DESENVOLVIMENTO FORMAL DE PROGRAMAS DE CONSULTA A BASES DE DADOS RELACIONAIS

Tese apresentada para CONCURSO de LIVRE DOCÊNCIA junto ao DEPARTAMENTO DE MATEMÁTICA APLICADA do INSTITUTO DE MATEMÁTICA E ESTATÍSTICA da UNIVERSIDADE DE SÃO PAULO

- São Paulo, março de 1978 -

— A todos os meus alunos,
especialmente àqueles que
tive o privilégio de orientar,
por terem dado um sentido humano
à minha atividade universitária.

— A meus pais e a
minha esposa, cuja
insistência para que
eu executasse este
trabalho foi um
fator decisivo na
sua elaboração.

ÍNDICE

INTRODUÇÃO v

CAP. 1 - DEDUÇÃO DE PROGRAMAS - EXEMPLO 1 1

CAP. 2 - DEDUÇÃO DE PROGRAMAS - EXEMPLO 2 15

CAP. 3 - CONSULTAS A BASES DE DADOS RELACIONAIS FORMULADAS EM CÁLCULO DE PREDICADOS. 38

3.1 - Introdução 38

3.2 - Exemplos de Consultas em ALPHA 41

3.3 - A Linguagem CONSULTOL. 44

3.3.1 - Introdução 44

3.3.2 - Sintaxe da Linguagem CONSULTOL 46

3.3.3 - Descrição Informal da Semântica da linguagem CONSULTOL. 49

3.4 - Exemplos de Consultas em CONSULTOL 51

CAP. 4 - ESQUEMAS PARA DEDUÇÃO DE PROGRAMAS DE CONSULTAS A BASES DE DADOS. 57

4.1 - Introdução 57

4.2 - Consultas sem Quantificadores. 57

4.3 - Consultas com Um Quantificador 67

4.4 - Consultas com Dois Quantificadores Existenciais. . . . 69

4.5 - Consultas com Quantificadores Existenciais 80

4.6 - Consultas com Quantificadores Universais 83

4.7 - Consultas com Ambos Quantificadores. 87

4.8 - Consultas com Negação de Quantificadores 89

CAP. 5 - APLICAÇÕES DOS ESQUEMAS DE TRANSFORMAÇÕES DE CONSULTAS A BASES DE DADOS 92

5.1 - Introdução 92

5.2 - Exemplo de Implementação do Operador de Escolha e Mudança nas estruturas de Dados. 94

CAP. 6 - CONCLUSÕES 104

APÊNDICE 107

1 - PROVA DE E2 107

2 - PROVA DE E8 109

3 - PROVA DE E10. 110

REFERÊNCIAS. 114

INDICES DOS ESQUEMAS DE TRANSFORMAÇÃO E DAS FORMAS DE CONSULTAS. . . 117

- ¤ -

INTRODUÇÃO

A programação de computadores nasceu como atividade improvisada por parte daqueles que projetaram as primeiras dessas máquinas. Muitos anos se passaram antes que se tornassem do domínio público as primeiras tentativas de tornar a programação uma atividade regulada por regras e princípios próprios. A introdução da programação estruturada, principalmente nas obras clássicas de Dijkstra [12] e Wirth [28], visou em parte o desenvolvimento de uma metodologia da programação, sintomaticamente denominada por Wirth de "Programação Sistemática" [29].No entanto, somente há bem pouco tempo é que se começou a formalizar o desenvolvimento de um programa, procurando-se aproximar a natureza dessa atividade da mesma que caracteriza a dedução de fórmulas matemáticas ou da física teórica [5], [6]. Uma tentativa anterior de formalizar a programação deu-se na pesquisa de métodos da assim chamada "Prova de Programas", isto é, técnicas para provar-se que um dado programa está correto (uma excelente introdução ao assunto pode ser encontrada em Manna [21], cap. 3). No entanto, em contraposição à dedução de programas, a verificação parte de um programa já pronto, e consegue em vários casos, a muitas penas e sem que haja uma regra geral a seguir, provar que um programa está de fato correto. Por outro lado, usando processos de dedução procura-se desenvolver programas corretos pela aplicação de transformações que

se assemelham às deduções da Lógica Matemática. Formula-se o algoritmo em uma linguagem "de muito alto nível", em geral em Cálculo de Predicados de 1ª ordem, e aplicam-se regras de transformação, previamente provadas, até que se obtenha um programa executável em computador. Na Universidade Técnica de Munique o Prof. F.L.Bauer formou uma equipe que está trabalhando intensamente nesse assunto, tendo como objetivo final a construção de um sistema computarizado para desenvolvimento de programas. Nesse sistema [4], um catálogo de transformações é armazenado no computador; o programador formula seu algoritmo em uma linguagem "de muito alto nível" [3] e, comandando o sistema, vai transformando essa formulação até atingir um programa executável. Nesse processo, a intuição do programador volta-se para a escolha dos esquemas de transformação que devem ser aplicados em cada passo, e eventualmente na dedução de novos esquemas. O nível do programa final pode ser até do mais baixo possível — o da linguagem de máquina, como o mostram Partsch e Pepper em [24].

Neste trabalho, demonstramos como a técnica de Munique pode ser aplicada no desenvolvimento de consultas a bases de dados. Para tanto, não podendo escapar à nossa vocação pedagógica, fizemos uma introdução àquela técnica, principalmente quanto ao formalismo desenvolvido por Gnatz [16], [17]. Essa introdução é feita nos capítulos 1 e 2. No primeiro, desenvolvemos um programa para um exemplo absolutamente trivial, o de determinar o maior e o menor de dois números inteiros dados. No segundo, damos um exemplo bem mais interessante e de alguma utilidade, desenvolvendo um programa para verificar se um número é ou não primo. No capítulo 3, fazemos uma introdução às consultas a bases de dados relacionais formuladas em cálculo de predicados, com referência

à linguagem ALPHA de Codd [9]. Para que possamos desenvolver esquemas de transformação para essas consultas, definimos uma linguagem própria, CONSULTOL, que se encaixa na que se compreende como "de muito alto nível", pela possibilidade de formulações em cálculo de predicados, uso de estruturas abstratas como conjuntos e relações, etc. O capítulo 4 é o mais importante deste trabalho: introduzimos e provamos vários esquemas de transformações para as consultas formuladas em CONSULTOL, dando versões recursivas e não-recursivas de algoritmos que denominamos de "semi-executáveis". Estes contém um operador de escolha que resta a ser implementado. Para mantermos a maneira didática de introduzir este assunto — já que se trata da primeira publicação no vernáculo sobre essa metodologia — introduzimos paulatinamente os esquemas, segundo o grau de complexidade das consultas. Um último esquema engloba todos os anteriores em generalidade, mas sua demonstração baseia-se numa generalização de esquemas anteriores. No capítulo 5 fazemos uma aplicação dos esquemas a uma consulta demonstrando os aspectos práticos do método. Nesse capítulo desenvolvemos o exemplo até a obtenção de um algoritmo "executável", mostrando duas implementações do operador de escolha, e a passagem das estruturas abstratas para estruturas existentes nas linguagens comuns de programação. Durante o desenvolvimento desse exemplo, introduzimos novos esquemas de transformação, os quais não são provados por penetrarem já num terreno semi-formal. No capítulo 6 fazemos breves considerações sobre a utilidade prática do método e sugerimos novas linhas de pesquisa baseadas neste trabalho. Todos os esquemas que não contém referências a terceiros são de nossa autoria, bem como, é claro, a linguagem CONSULTOL.

Gostaríamos aqui de agradecer profundamente ao Dr.

R. Gnatz, da T.U.München, que nos convidou em 1977 a esta-
giar junto ao seu grupo, que nos recebeu tão carinhosamente
e que, pela sua dedicação em nos introduzir ao método de
transformações ainda parcamente divulgado naquela época,pos
sibilitou que desenvolvessemos interesse pelo assunto, cul-
minando nos resultados deste trabalho.

- o -

CAPÍTULO 1

DEDUÇÃO DE PROGRAMAS — EXEMPLO 1

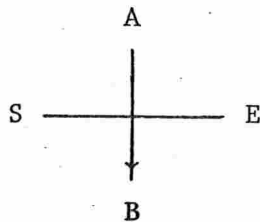
Neste capítulo exemplificaremos o método de dedução de programas corretos atualmente empregados pela equipe do Prof. F.L.Bauer, de Munique, principalmente o formalismo empregado por R. Gnatz. Um resumo desse método pode ser encontrado em [16] e seus detalhes em [17], principalmente quanto às deduções formais precisas. O exemplo que desenvolvemos abaixo foi, propositadamente, centrado em um problema absolutamente trivial do ponto de vista algorítmico ou de programação de computadores. Isso se deveu ao fato de termos querido, com isso, descrever os processos de dedução do algoritmo correspondente a esse problema em todo o detalhe; um problema complexo poderia desviar a atenção do formalismo em si. No capítulo seguinte desenvolveremos outro exemplo, aplicando então algumas passagens de maneira mais sucinta, demonstrando a aplicação prática do método.

Para deduzir-se formalmente um programa segundo o método em questão, usaremos o seguinte formalismo introduzido por Gnatz [17].

Um problema é descrito por um triplo ordenado (C,P,A), onde A é um programa, C um predicado constituído de

condições válida antes da execução de A e P um predicado que deve ser satisfeito após a execução de A. (C,P,A) é admissível se a proposição C $\Longrightarrow$ P(A), onde "$\Longrightarrow$" é a implicação lógica, pode ser provada, isto é, dentro da teoria em questão $\vdash$ C $\Longrightarrow$ P(A), isto é, C $\Longrightarrow$ P(A) é um teorema.

Note-se que esta notação difere daquela introduzida por Floyd [15] e desenvolvida por Hoare [19]; ela permite a aplicação direta da lógica tradicional, em particular da formulação de teorias e do cálculo de predicados. O programa A será, como veremos, desenvolvido por meio da aplicação de regras de transformação previamente provadas, também denominadas de esquemas. A notação que usaremos para as transformações de um programa A em um programa B será a seguinte:



Essa transformação deve ser entendida da seguinte maneira: o programa A pode ser substituído pelo programa B no caso das condições S serem preenchidas, tendo se aplicado o esquema (ou conjunto de esquemas) E de transformação. Assim, suponhamos que E seja um esquema válido dentro da teoria; se (C,P,A) for um triplo admissível para um determinado problema, isto é, se C $\Longrightarrow$ P(A) puder ser provado, então se as condições S forem verdadeiras, aplicando-se o esquema E pode-se demonstrar que C $\Longrightarrow$ P(B).

Seja o seguinte

Problema dados 2 números inteiros, desenvolver um programa para determinar qual é o menor e qual é o maior.

Temos então:

C: são dados dois números inteiros p e q.

Em termos de linguagens de programação, poderíamos escrever

C: <u>inteiro</u> p,
<u>inteiro</u> q.

Com isso especificamos que p e q pertencem ao conjunto dos inteiros. Leia-se a vírgula como a conjunção lógica "<u>e</u>".

Com estas condições, o programador poderia formalizar o problema da seguinte maneira: se o resultado procurado é o par (x,y), ele deve satisfazer a

$$P(x,y) \equiv p\leq q \wedge x=p \wedge y=q \vee p>q \wedge x=q \wedge y=p,$$

isto é, o problema é determinar o par ordenado (x,y) que satisfaz o predicado P (neste caso uma sentença do cálculo proposicional). Nesse par o primeiro elemento será o menor dos dois dados e o outro, o maior.

A existência de uma solução para o problema é garantida pela seguinte asserção:

$$C \Longrightarrow \exists\ (\underline{\text{inteiro}}\ x,\ \underline{\text{inteiro}}\ y) : P(x,y)$$

isto é (daqui em diante usaremos abreviaturas onde elas forem óbvias, aliadas à sintaxe comum às linguagens de programação):

$$C \Longrightarrow \exists(\underline{\text{int}}\ x,y) : p\leq q \wedge x=p \wedge y=q \vee p>q \wedge x=q \wedge y=p$$

Esta asserção pode ser deduzida da teoria dos números:

dados dois inteiros p e q, $p\leq q \vee p>q$ é uma tautologia;por

outro lado, dados dois números sempre existem dois números iguais,respectivamente a cada um deles – que são os próprios dados.

Temos, então, um ponto de partida para a dedução do programa:

(1) $\vdash C \Longrightarrow \exists(\underline{int}\ x,y): p \leq q \wedge x = p \wedge y = q \vee p > q \wedge x = q \wedge y = p$

Neste ponto empregamos uma peça fundamental no método, uma versão do Axioma da Escolha devida a Hilbert-Barnays [17]:

$$\vdash \exists x{:}P(x) \Longrightarrow P(\eta x{:}P(x))$$

onde P é um predicado e η um operador que denominaremos de "Operador de Escolha"; o argumento de P no lado direito da implicação deve ser lido da seguinte maneira: "algum x tal que P(x)". Intuitivamente, a existência de um x permite-nos afirmar que para algum x (que logicamente deverá satisfazer P), P será verdadeiro. Temos, então, uma nova versão de nossa proposição:

(2) $\vdash C \Longrightarrow P(\eta\ \underline{int}\ x,y{:}\ p \leq q \wedge x = p \wedge y = q \vee p > q \wedge x = q \wedge y = p)$

baseada na transitividade de $\Longrightarrow: A \Rightarrow B,\ B \Rightarrow C \vdash A \Rightarrow C$.

Em termos de programas, (2) poderia ter sido escrita imediatamente a partir de (1) pela aplicação do seguinte esquema de transformação:

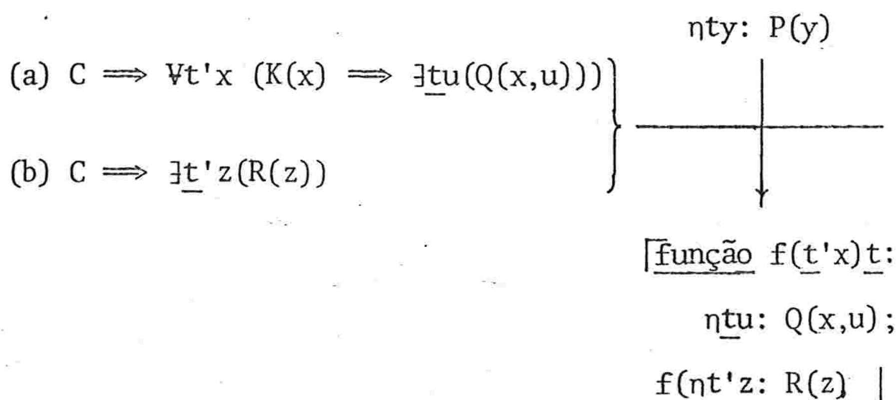
(esquema E1):

$$\vdash C \Longrightarrow \exists \underline{t}x{:}P(x) \xrightarrow{\ \varepsilon\ } \eta \underline{t}x{:}\ P(x)$$

onde ε é o programa vazio, t um tipo (eventualmente composto), e x uma variável (eventualmente composta). A prova desse esquema segue passos exatamente análogos à dedução de (2), lembrando-se que, com o formalismo introduzido inicialmente, a conclusão (parte inferior) do esquema deve ser escrita sem o predicado P, pois é um programa cujos resultados (no caso x), devem satisfazer aquele predicado.

O passo seguinte é a transformação do programa, assim obtido, na declaração e chamada de um procedimento ("procedure") (no caso uma função que retorna valores) com o seguinte esquema (V. [17]):

(E2)



onde $R(z) \equiv (C \Longrightarrow K(z)) \wedge \forall tu(Q(z,u) \Longrightarrow P(u))$.

Usamos a notação ⌈...⌋ em lugar dos delimitadores de bloco tradicionais begin...end, para maior clareza, bem como suas variações ⌈...⌋, ⌈...⌋, etc.; como em ALGOL-68 [26] e ALGOL-W [30], blocos podem retornar valores de expressões; usamos a seguinte sintaxe para procedimentos do tipo função:

função <nome da função> (<lista de parâmetros formais com seus tipos>)
<lista de tipos dos resultados>:<corpo da função>

A demonstração desse esquema é bastante complexa; para não sobrecarregarmos este capítulo, apresentamo-la no

apêndice 1. Intuitivamente, pode-se entender a sua validade considerando-se que, pela definição de um procedimento, a sua chamada equivale a uma substituição do corpo do procedimento no lugar da chamada, colocando-se em cada ocorrência de um parâmetro formal o correspondente parâmetro atual (isto é, o da chamada). Em nosso caso, em lugar de x colocamos o z escolhido na chamada, e que satisfaz o predicado R; a existência de pelo menos um z nessas condições é garantida por (b). Por outro lado, para esse z, R garante que P será satisfeito para qualquer u; (a) nos garante que, pelo menos para o z determinado em (b), há um u que satisfaz Q e, portanto, P. Note-se que, no fundo, houve um desdobramento de P, através de Q, em um predicado de duas variáveis, o que permite, em última instância, a chamada do procedimento independente de y e a colocação de alguns parâmetros de P diferentes de y como variáveis no corpo do procedimento, isto é, no caso, x de Q(x,u).

Para aplicarmos esse esquema, é necessário achar-se os predicados K e Q convenientes; nesse ato é que a intuição do programador entrará em ação. Em nosso caso, é mais ou menos evidente que os parâmetros da função deverão ser os dados do problema. Escolhemos então, para o parâmetro x do esquema, o par (p,q); lembrando que na definição inicial de P, P(u) = P(x,y), escolhemos

$$Q(p,q,x,y) \equiv p\leq q \wedge x=p \wedge y=q \vee p>q \wedge x=q \wedge y=p,$$

isto é, Q equivale essencialmente a P a menos da "parametrização" de p e q que, para P, funcionavam como "variáveis globais".

Por outro lado, tomaremos

$$K(p,q) \equiv \underline{\text{int}}\ p,\ \underline{\text{int}}\ q$$

Com essa escolha, esses dois predicados satisfazem trivialmente às condições (a) e (b) do esquema E2. De fato, recordando que, em nosso exemplo, $C \equiv$ int p, int q, temos $C \equiv K$. Neste caso, (a) equivale a (1) e é portanto satisfeita; quanto a (b) a primeira parte de R é verdadeira para qualquer z (pois $C \equiv K$) e evidentemente para o Q acima, $\forall$(int x, int y): $(Q(p,q,x,y) \Longrightarrow P(x,y))$. Assim, qualquer par (p,q) satisfaz R.

Portanto (2) transforma-se pela aplicação de E2, em:

(3) $\vdash C \Longrightarrow P(\lceil$função f(int p',q')int,int:

η int x,y: $p' \leq q' \wedge x=p' \wedge y=q' \vee p>q' \wedge x=q' \wedge y=p'$;

f(p,q)$\rfloor$)

Note-se que o programa entre os delimitadores de blocos já é um programa que chamaremos de "semi-executável" no sentido computacional. Ele tornar-se-á "executável" se o operador de escolha η for implementado por meio de uma rotina qualquer. Por exemplo, poderíamos construir um gerador de pares de inteiros usando um processo semelhante à diagonalização: (0,0), (1,0), (0,1), (-1,0), (0,-1), (2,0), (1,1), (0,2), (-2,0), (-1,-1), (0,-2), etc. Para cada par, testaríamos o predicado $p' \leq q' \wedge x = p'$... e o primeiro par que satisfizer ao predicado será o resultado da função e, portanto, do programa (no caso a solução é também única). Examinando-se melhor o programa de (3), pode-se também concluir que uma implementação eficiente de η seria testar os pares (p,q) e (q,p) devido às condições $x = p' \wedge y = q'$ e $x = q' \wedge y = p'$ que, uma ou outra, deverão ser satisfeitas. No entanto, podemos continuar nossa síntese um pouco mais a fim de deduzirmos um programa ainda mais

eficiente. Para isso, apliquemos o seguinte esquema (cf.[17]):

(E3)

$$\frac{\underline{\eta t}x:\ (P_1 \wedge x = A_1 \vee P_2 \wedge x = A_2 \vee \dots \vee P_m \wedge x = A_m)}{\underline{se}\ P_1 \rightarrow A_1\ \square\ P_2 \rightarrow A_2\ \dots\ \square\ P_m \rightarrow A_m}\quad x \text{ não livre em } P_i \text{ e em } A_i,\ i=1,2,\dots,m$$

onde P_i são predicados, A_i expressões com valores do tipo $\underline{t}$ e na parte inferior do esquema temos um "guarded command" como definido por Dijkstra em [14]. É retornado como seu valor o valor da expressão A_i para a qual P_i é verdadeiro; se P_i e P_j são verdadeiros com $i \neq j$, será relacionado um dos valores A_i ou A_j como o valor do comando neste seu processamento. Numa outra execução com as mesmas condições, o mesmo ou outro valor podem ser selecionados. Assim, o comando pode ser não-determinístico. Dijkstra utiliza A_i como uma sequência de comandos de execução da esquerda para a direita;usando a forma de expressão condicional do ALGOL 60 [23],temos a extensão acima válida para expressões, que poderíamos denominar de "guarded expression". No caso de $m = 2$ e os predicados P_1 e P_2 serem mutuamente exclusivos, isto é, no caso em que $P_1 \Longrightarrow \neg P_2$ (e portanto $P_2 \Longrightarrow \neg P_1$) é uma tautologia, temos o tradicional comando <u>if</u>...<u>then</u>...<u>else</u>... determinístico, isto é, obtemos o seguinte esquema:

(E4)

$$\frac{\eta\underline{t}x:\ (P\wedge x = A_1 \vee \neg P\wedge x = A_2)}{\underline{se}\ P\ \underline{então}\ A_1\ \underline{senão}\ A_2}\quad \text{x não livre em } P, A_1 \text{ e } A_2$$

Para podermos aplicar E4 à versão (3) do nosso programa, basta fazermos uma transformação de notação, isto é, a conjunção $x = p' \wedge y = q'$ será substituída por $(x,y) = (p',q')$, ou seja a igualdade de dois pares ordenados. A equivalência desses dois termos é trivialmente demonstrada por meio de uma Tabela de Valores ("truth table"). Obtemos, então

(4) $\vdash C \Longrightarrow P(\ulcorner$ <u>função</u> f (<u>int</u> p',q')<u>int</u>, <u>int</u>:

η <u>int</u> x,y: $p' \leq q' \wedge (x,y) = (p',q') \vee$

$p' > q' \wedge (x,y) = (q',p')$;

f(p,q)$\lrcorner$)

Agora podemos aplicar E4 pois $p' \leq q' \vee p' > q'$ é uma tautologia e os termos são mutuamente exclusivos.

(5) $\vdash C \Longrightarrow P(\ulcorner$ função f (<u>int</u> p',q') <u>int</u>, <u>int</u>:

<u>se</u> $p' \leq q'$ <u>então</u> (p',q') <u>senão</u> (q',p');

f(p,q)$\lrcorner$)

O programa pode ser considerado como em sua forma quase ideal, isto é, ótima. A única restrição que um programador poderia fazer a essa versão seria a de que não é necessário declarar-se e chamar-se um procedimento para efetuar a computação requerida; afinal, a chamada de um procedimento sempre implica em perda de certo tempo de execução... Para satisfazer este perfeccionista, podemos aplicar

um esquema que Burstal e Dalington denominaram em [5] de "unfolding" (traduziremos por "expansão") (V. também [6], uma versão mais elaborada de [5]). Neste trabalho eles desenvolvem um sistema estritamente funcional, sem a notação usual de linguagens de programação. Usaremos, por isso, um esquema semelhante descrito por Bauer et al. [2]. Esse esquema engloba a expansão de chamadas de procedimentos bem como a transformação contrária, isto é, a "contração" ("folding"):

(E5) ⌈<u>função</u> f (<u>t</u>x)<u>t</u>': S;
... f(E) ...⌋

expansão ↑ ↓ contração

⌈<u>função</u> f (<u>t</u>x)<u>t</u>': S;
... S_E^x ...⌋

onde E e S denotam expressões e S_E^x a expressão S reescrita substituindo-se todas as ocorrências de x pela expressão E (S_E^x substitui a ocorrência de f(E)). Esse esquema decorre diretamente da definição de procedimentos sem efeitos colaterais (v.p. ex. [23], item 5.4.3, e [26], item 5.4.3.2).

Aplicando E5 a (5) obtemos:

(6) ⊢ C ⟹ P(⌈<u>função</u> f (<u>int</u> p',q') <u>int</u>, <u>int</u>:
<u>se</u> p'≤q' <u>então</u> (p',q') <u>senão</u> (q',p');
<u>se</u> p≤q <u>então</u> (p,q) <u>senão</u> (q,p)⌋,

Examinando (6), verificamos que a declaração da função é superflua. Pulando a formalização trivial da transformação de um trecho vazio de programa em um procedimento não

chamado em nenhum local desse programa evice-versa,obtemos:

(7) $\vdash C \Longrightarrow P(\underline{se}\ p\leq q\ \underline{então}\ (p,q)\ \underline{senão}\ (q,p))$

O programa sintetizado é, portanto,

$\underline{se}\ p\leq q\ \underline{então}\ (p,q)\ \underline{senão}\ (q,p)$ #

isto é, se a pré-condição C (isto é, int p, int q é verdadeira, esse programa satisfaz o predicado P como enunciado inicialmente. Se esse predicado exprime corretamente as condições do problema, o programa é uma versão correta da solução do problema,isto é, acha o menor e o maior de dois inteiros p e q.

Observações:

1) Repetindo nossa observação inicial,o exemplo é extremamente simples, mas tivemos em mente,ao desenvolvê-lo,mostrar em todo o detalhe a aplicação do método. Note-se que foram empregados alguns esquemas extremamente poderosos, que na verdade prestam-se a casos bem mais complexos. Como nosso exemplo foi muito simples, pode-se ter a impressão de que o método é complicado demais.Veremos no próximo capítulo,em um exemplo bem mais interessante,que algumas passagens podem ser abreviadas sem prejuízo da "corretude", tornando o método viável.

2) Uma questão fundamental é a da formulação das pré-condições C e do predicado inicial (P(x,y)) que exprime a propriedade dos resultados do programa. Ambos,nesse método são expressos usando-se todo o poder do cálculo de predicados de 1ª ordem, aliado à notação comum de linguagens de programação "tipo ALGOL". Em nosso caso, um matemático poderia ter formulado P da seguinte maneira:

$$P(x,y) = p\leq q \wedge x=p \wedge y=q \vee p\geq q \wedge x=q \wedge y=p$$

A dedução teria sido exatamente a mesma até o teorema (4). (5) deveria então ser

(5') $\vdash C \Rightarrow P(\ulcorner$ <u>função</u> f (<u>int</u> p',g')<u>int</u>, <u>int</u>:

<u>se</u> $p' \leq q' \longrightarrow (p',q')$

$\square$ $p' \geq q' \longrightarrow (q',p')$;

f(p,q)$\lrcorner$)

isto é, conservaríamos a formulação não-determinística de Dijkstra. Obteríamos finalmente o programa

<u>se</u> $p \leq q \longrightarrow (p,q)$

$\square$ $p \geq q \longrightarrow (q,p)$

que é praticamente o resultado dado por Dijkstra em [14], com a diferença que ele calcula apenas o maior dos dois números e usa o comando de atribuição. Para chegarmos à sua formulação, achemos somente o máximo:

<u>se</u> $p \leq q \longrightarrow p$

$\square$ $p \geq q \longrightarrow q$

e façamos uma atribuição dessa expressão a uma variável:

$m \longleftarrow$ <u>se</u> $p \leq q \longrightarrow p$

$\square$ $p \geq q \longrightarrow q$

Ora, essa forma pode ser transformada na forma seguinte empregando-se a definição de "guarded command" e sua extensão para "guarded expression":

$\underline{se}$ $p \leq q \longrightarrow m \leftarrow p$

$\square$ $p \geq q \longrightarrow m \leftarrow q$

obtendo, agora, exatamente a forma apresentada por Dijkstra, [14,pg. 456]. Evidentemente surge a seguinte pergunta: qual a diferença entre os dois métodos? O método de Gnatz, por nós exposto, resume-se à aplicação estritamente formal de esquemas de transformação previamente provados, obtendo-se um processo dedutivo. No caso de Dijkstra, procura-se simplesmente completar os "guards", não se recorrendo a um catálogo de esquemas já existente. De certa maneira, poderíamos dizer que o processo de Dijkstra inclui uso de esquemas de transformação não-formais, isto é, intuitivos.

3) Examinando o método acima exposto, podemos observar os seguintes passos na solução de um problema:

Passo 1: Formulação dos predicados que se aplicam aos dados do problema, isto é, estabelecimento das pré-condições C.

Passo 2: Formulação dos predicados P que devem ser satisfeitos pelos resultados esperados.

Passo 3: Prova da existência de uma solução para o problema, como consequência da implicação em que as pré-condições C constituem o antecedente.

Passo 4 e seguintes: aplicação dos esquemas de transformações até a obtenção de um algoritmo "executável" (no sentido computacional); algumas transformações podem ser feitas com o intuito de se aumentar a eficiência do programa.

Note-se que o resultado do Passo 3 permite a aplicação do esquema E1, obtendo-se uma primeira versão do programa. A aplicação de E2 pode ser, em geral, feita intuiti-

vamente; no fundo o que se deseja é uma parametrização dos predicados obtidos no Passo 2, e no caso em que, como no exemplo visto, $K \equiv C$ e Q é apenas uma versão parametrizada de P, a aplicação de E2 torna-se trivial.

- ¤ -

CAPÍTULO 2

DEDUÇÃO DE PROGRAMAS — EXEMPLO 2

Neste capítulo veremos um exemplo bem mais interessante, do ponto de vista computacional. Além disso, em lugar do predicado a ser satisfeito pelos resultados ser uma simples proposição (sentença do cálculo proposicional) como no exemplo do capítulo anterior, teremos um predicado de 1ª ordem propriamente dito. Finalmente, o procedimento resultante será recursivo, e teremos então oportunidade de aplicar um esquema de eliminação da recursão.

Problema: dado um número inteiro maior do que 1, verificar se ele é primo.

Temos:

$$C: \underline{\text{int}}\ p,$$
$$p>1.$$

A condição de p ser ou não ser primo pode ser descrita pelo predicado

$$\neg\ \exists\ \underline{\text{int}}\ q(q<p \wedge q>1 \wedge p\ \underline{\text{mod}}\ q=0)$$

onde p mod q é o resto da divisão (inteira) de p por q.

Suponhamos que lógico x (ou log x) declare x como sendo uma variável lógica, isto é, x assume valores no conjunto {verdadeiro, falso} ou {ver, fal}. Assim sendo, podemos dizer que o resultado procurado (e a ser calculado pelo programa) é o valor da variável x onde

$$x = \neg\exists\ \underline{\text{int}}\ q(q<p \wedge q>1 \wedge p\ \underline{\text{mod}}\ q = 0)$$

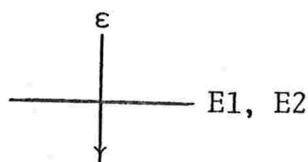
Portanto, usando a notação do capítulo anterior,

$$P(x) \equiv x = \neg\exists\ \underline{\text{int}}\ q(q<p \wedge q>1 \wedge p\ \underline{\text{mod}}\ q = 0)$$

Evidentemente o predicado $\neg\exists$ int q... é verdadeiro ou falso: dado um inteiro p qualquer, podemos testar todos os inteiros q menores do que ele e maiores do que 1,pois o número de q's é finito. Portanto

$$\vdash C \Longrightarrow \exists\ \underline{\text{log}}\ x(P(x))$$

o que nos garante a existência de solução para o problema e nos fornece o ponto de partida para a dedução do programa procurado. Aplicando os esquemas segundo o formalismo introduzido no capítulo anterior, temos:



⌈ função primo (int p') log:

(1) $\quad \eta\ \underline{\text{log}}\ x$: $x = \neg\exists\ \underline{\text{int}}\ q$: $q<p' \wedge q>1 \wedge p'\ \underline{\text{mod}}\ q = 0$;

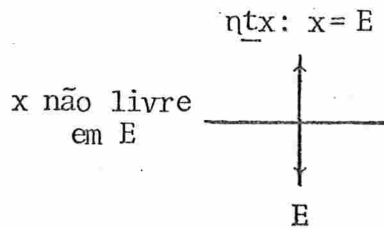
primo (p)⌋

Para a aplicação de E2 tomamos $K(p) \equiv C$; $Q(p,x) \equiv P(x)$;

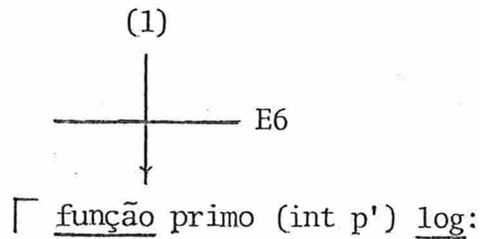
em lugar de $f(\eta\underline{t}'z: R(z))$ podemos colocar primo (p) pois R(p) é válido para todo p devido à identidade de Q e P e de K e C.

Observando o procedimento "primo", podemos notar que a variável x é supérflua; aliás, podemos aplicar o esquema trivial

(E6)



obtendo



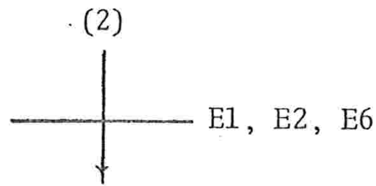
(2)

⌈ <u>função</u> primo (int p') <u>log</u>:

$\neg\exists$ <u>int</u> q $(q<p' \wedge q>1 \wedge p'$ <u>mod</u> $q=0)$;

primo (p) ⌋

Note-se que, dado o predicado inicial (logo após as pré-condições C), poderíamos ter escrito (2) imediatamente por intuição, tratando-se, em essência, da parametrização de P(x) e a sua transformação em <u>função</u>. Para exemplificar esse processo, e evidentemente tendendo a uma forma mais adequada para as transformações subsequentes, vamos isolar o predicado a partir do quantificador como um procedimento à parte:



(3)

⌈ <u>função</u> primo (<u>int</u> p') <u>log</u>:

‖⌈ <u>função</u> nprimo (<u>int</u> p'') <u>log</u>:

$\exists$ <u>int</u> q(q<p''∧q>1∧p'' <u>mod</u> q = 0);

$\neg$ nprimo (p') ⌋‖;

primo (p) ⌋

Observe-se que, tomando em E2 K(x)≡C e Q(x,u)≡P(x), tudo se passa como se o predicado inicial P(x) fosse expresso como um procedimento, cujos parâmetros formais são as variáveis declaradas nas pré-condições C, adicionando-se ainda uma chamada a esse procedimento usando-se como parâmetros atuais as mesmas variáveis de C (temos usado identificadores diferentes para maior clareza). Com isso, o desenvolvimento até este ponto foi relativamente automático. Agora é que a intuição do programador deverá atuar decisivamente. O raciocínio R que poderia ser seguido seria, por exemplo: "É preciso produzir a variação dos valores de q desde 2 até p-1. Para isso, precisamos de mais uma variável, livre em relação ao quantificador existencial. Como introduzir mais uma variável? Uma maneira seria substituir p'' no termo q < p'' por uma variável q'' que ocorresse como parâmetro formal de um novo procedimento, o qual além dela teria p'' como outro parametro formal."

Para concretizarmos o raciocínio R, vamos empregar o seguinte esquema exposto por Bauer et al. [2], ao qual acrescentamos uma condição que usa a operação de substitui-

ção descrita no capítulo anterior (V. E5).

(E7) <u>função</u> f (<u>t</u>x)<u>t</u>': S

$S'^{y}_{y_0} \equiv S$ ——|——

⌈ <u>função</u> f(<u>t</u>x)<u>t</u>': g(x,y_0);

<u>função</u> g(<u>t</u>x,<u>t</u>''y)<u>t</u>':S' ⌋

isto é, substituímos a expressão S pela chamada de uma outra função g com um parâmetro y a mais do que f, de tal maneira que para o valor inicial y_0 (que ocorre como parâmetro atual na chamada de g) de y o corpo S' de g seja idêntico a S. A demonstração do esquema pode ser feito em cada caso por expansão e contração, aplicando-se o esquema E5 do capítulo anterior.

Em nosso caso, teremos, usando o raciocínio R:

(3)

——|—— E7

⌈ função primo (<u>int</u> p') <u>log</u>:

⌈⌈ <u>função</u> nprimo (<u>int</u> p'') <u>log</u>: $nprimo_2$ (p'',p'');

(4) <u>função</u> $nprimo_2$ (<u>int</u> p''',q'''):

$\exists$ <u>int</u> q(q<q'''∧q>1∧p''' <u>mod</u> q = 0);

¬ nprimo (p') ⌋⌋;

primo (p) ⌋

A condição do esquema E7 é evidentemente preenchi-

da.

Agora poderemos tratar da variação de q''' segundo o raciocínio R. Já que o valor inicial de q''' é p'' (e portanto p', e portanto p), uma possibilidade seria de variar q''' em ordem descendente de grandeza: p, p-1,... Essa variação irá, pelo racicínio R, parar em 2, quando o teste especificado pelo predicado não deverá ser mais efetuado. Vamos então introduzir o teste de parada. Para isso, basta introduzirmos no predicado a tautologia $q''' = 2 \vee q''' \neq 2$, e depois desmembrá-la nos dois termos de que ela se compõe. É evidente que a introdução de uma tautologia numa sequência de conjunções não altera o valor da proposição. No entanto, um esquema descrito por Gnatz [17] ajuda-nos a fazer esse passo fundamental (V. prova no Apêndice):

(E8)

$\eta \underline{t}x$: P(x)

(a) $\vdash C \Longrightarrow \exists\ \underline{t}x(Q(x))$

(b) $\vdash C \Longrightarrow (Q(x) \Longrightarrow P(x))$

$\eta \underline{t}x$: Q(x)

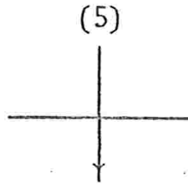
Aplicando novamente E6 (agora em sentido contrário), poderíamos introduzir o operador η e uma variável x, definindo-se um novo predicado igual ao anterior precedido de x=. Depois de aplicar E8, poderíamos eliminar η e x novamente por meio de E6. O resultado final seria:

(4)

(5)

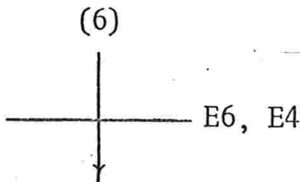
```
⌈função primo (int p') log:
    ⟦⌈função nprimo (int p'') log: nprimo₂ (p'',p'');
      função nprimo₂ (int p''',q'''):
          ∃ int q((q'''=2vq'''≠2)∧q<q'''vq>1∧p mod q= 0);
      ⌉ nprimo (p')⌋⟧;
primo (p)⌋
```

Por meio de propriedades do cálculo proposicional e do cálculo de predicados, aliados a E8 chegamos a:




```
⌈função primo (int p') log:
     ⌈⌈função nprimo (int p'') log: nprimo₂ (p'',p'');
      função nprimo₂ (int p''',q'''):
(6)            q''' = 2∧∃int q(q<q'''∧q>1∧p''' mod q = 0)∨
                     q'''≠2∧∃int q (q<q'''∧q>1∧p''' mod q = 0);
     ⌉ nprimo (p')⌋⌋;
  primo (p)⌋
```

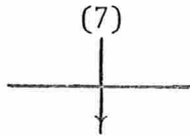
Novamente aplicando-se E6 como no capítulo anterior, chegar-se-ia à foram adequada para a utilização de E4, que nos forneceria




```
⌈função primo (int p') log:
     ⌈⌈função nprimo (int p'') log: nprimo₂ (p'',p'');
      função nprimo₂ (int p''',q'''):
(7)            se q''' = 2 então ∃ int q (q<q'''∧q>1∧p''' mod q = 0)
                           senão ∃ int q (q<q'''∧ >1∧p''' mod q = 0);
     ⌉ nprimo (p')⌋⌋;
  primo (p)⌋
```

Ora, ∃ int q ($q<q''' \wedge q>1 \wedge p'''$ mod $q = 0$) é falso se $q''' = 2$ pois para satisfazer $q < q'''$ teríamos $q < 2$ o que contradiz com o termo $q < 1$. Para exprimirmos explicitamente a variação de q''', vamos introduzir uma nova tautologia:

(7)



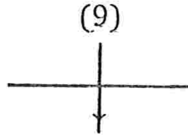
```
    ⌈função primo (int p') log:
         ‖⌈função nprimo (int p'') log: nprimo₂ (p'',p'');
          função nprimo₂(int p''',q'''):
(8)             se q'''=2 então falso
                          senão ∃ int q((q=q'''-1∨q≠q'''-1)∧q<q'''∧
                                      q>1∧p''' mod q =0);
         ⌉ nprimo (p')⌋‖;
     primo (p)⌋
```

```
    ⌈função primo (int p') log:
         ‖⌈função nprimo (int p'') log: nprimo₂(p'',p'');
          função nprimo₂(int p''',q''') log:
(9)             se q'''=2 então falso
                          senão ∃ int q(q=q'''-1∧q<q'''∧q>1∧
                                                p''' mod q =0)∨
                                ∃ int q(q≠q'''-1∧q<q'''∧q>1∧
                                                p''' mod q =0);
         ⌉ nprimo (p')⌋‖;
     primo (p)⌋
```

Mas

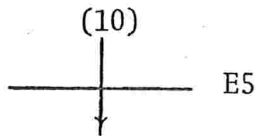
$$q = q'''-1 \wedge q<q''' \equiv q = q'''-1 \text{ e se } q''' \neq 2$$

então $q=q'''-1 \wedge q>1 \equiv q=q'''-1$ e, finalmente $q=q'''-1 \wedge p'''$ mod $q=0 \equiv p'''$ mod$(q'''-1)=0$.Por outro lado $q \neq q'''-1 \wedge q<q''' \equiv q<q'''-1$. Obtemos, então:




```
(10)  ⌈função primo (int p') log:
         ⟦⌈função nprimo (int p'') log: nprimo (p'',p'');
          função nprimo₂(int p''',q''') log:
               se q'''=2 então falso
                         senão p''' mod(q'''-1)=0∨
                                   ∃ int q(q<q'''-1∧q>1∧p'''mod q=0);
         ⌉ nprimo (p')⌋⟧;
      primo (p)⌋
```

Aqui aplicamos uma técnica fundamental do método: observando que $\exists$ int q: $q<q'''-1 \wedge q>1 \wedge p'''$mod $q=0$ é exatamente o corpo da função $nprimo_2$ como declarada em (4) a menos de $q'''-1$ de (12) que aparece no lugar de q''' de (4), podemos aplicar a contração de E5 obtendo $nprimo_2(q'''-1)$ no lugar do predicado em questão:

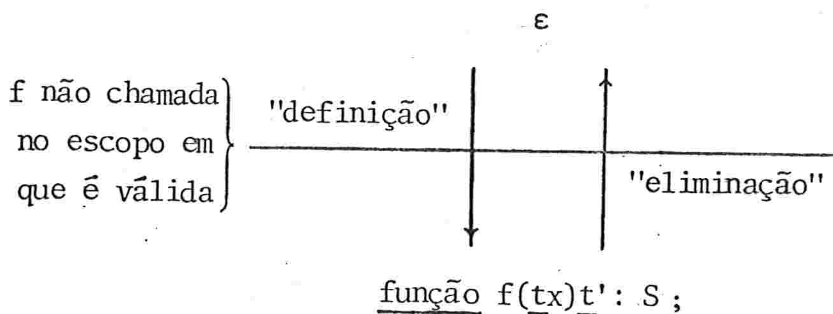



```
(13)  ⌈função primo (int p') log:
         ⟦⌈função nprimo (int p'') log: nprimo (p'',p'');
          função nprimo₂(int p''',q''') log:
               se q'''=2 então falso
                         senãop'''mod(q'''-1)=0∨nprimo₂(p''',q'''-1);
         ⌉ nprimo (p')⌋⟧;
      primo (p)⌋
```

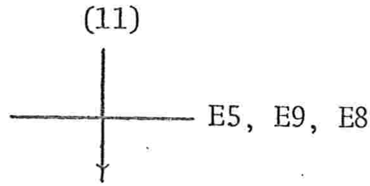
Obtivemos, assim, uma versão "executável" do problema: ela poderia ser facilmente convertida para uma das linguagens de programação usuais, desde que ela tivesse a possibilidade de processar procedimentos recursivos. Note-se também que a "parada" da recursão é garantida pelo fato de que o mínimo valor que assume q''' (isto é, p) é 2; para este caso não há chamada recursiva e para todos os outros (p>2, já que p>1 era uma das pré-condições C), a cada chamada recursiva o valor de q''' diminui de 1.

Poderíamos parar o processo de dedução neste ponto. No entanto, vamos continuá-lo, gerando novas versões mais eficientes. Em primeiro lugar, podemos observar que o procedimento nprimo e sua chamada são supérfluos. Para eliminá-lo, usemos a expansão de E5 para o comando nprimo (p') o procedimento nprimo passa a não ser chamado em nenhum ponto do programa, podendo então ser eliminado. Formalmente, essa eliminação é garantida pelo esquema seguinte, introduzido por Bauer et al. em [2] apenas no sentido de "definição" mas que evidentemente é válido também no outro sentido que chamaremos de "eliminação" (recorde-se que ε é um comando vazio):

(E9)



Finalmente fazemos uma terceira transformação, mudando os identificadores dos parâmetros formais de $nprimo_2$ (garantida por E8), obtendo-se:




```
         ⌈função primo (int p') log:
               ⟦⌈função nprimo₂(int p'',q''') log:
                      se q'' = 2 então falso
(12)
                               senão p'mod(q''-1)=0vnprimo₂(p'',q''-1);
                 ¬ nprimo₂(p',p')⟧;
            primo (p)⌋
```

Note-se que a eliminação de nprimo poderia ter sido feita na versão (4); não o fizemos para adiarmos a otimização do programa para esta fase final, deixando assim, em nosso entender, de sobrecarregar a dedução.

Como passo seguinte, mudemos a natureza de nprimo, que devolve o resultado falso se um número não é primo, e verdadeiro caso contrário, para o oposto, eliminando-se assim a negação à frente de sua chamada. Para isso desenvolvemos o esquema seguinte (cuja prova apresentamos no Apêndice):

```
(E10)            ⌈função f(tx)t': S;
                  :
                  ¬ f(y)
                  :
                        ⌋
                      ⟷
                 ⌈função f(tx)t': S;
                  função f'(tx)t': ¬ S^{f'}_{¬f'};
                  :
                  f'(y)
                  :     ⌋
```

Em nosso caso, $f(y) \equiv nprimo_2(p',p')$; tomemos $f' \equiv$ $\equiv primo_2$ e como $nprimo_2$ torna-se supérfluo, podemos eliminá-lo:

(12)

$\downarrow$ E9, E10

⌈ função primo (int p') log:
  ⌈⌈ função $primo_2$(int p'',q'') log:
(14)   ¬ se q''=2 então falso
     senão p'' mod (q''-1)=0 ∨ ¬ $primo_2$(p'',q''-1);
  $primo_2$(p',p') ⌋⌋;
 primo (p) ⌋

Pela definição de expressão condicional, temos:

(E11)

f(se E então E_1 senão E_2)

$\updownarrow$

se E então f(E_1) senão f(E_2)

Aplicando E11 com f = ¬ ao corpo de $primo_2$ e efetuando as operações do cálculo proposicional resultantes, expandindo a chamada primo(p), e eliminando a declaração de primo que é agora supérfluo, e finalmente mudando os identificadores dos parâmetros de $primo_2$ de p'',q'' para p',q' temos:

(13)

$\downarrow$

⌈ função $primo_2$(int p',q') log:
(14) se q'=2 então verdadeiro
   senão p' mod (q'-1) ≠ 0 ∧ $primo_2$(p',q'-1);
 $primo_2$(p,p) ⌋

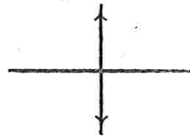
Em qualquer sistema de computação, a chamada recursiva de procedimentos consome um tempo considerável. Mesmo em máquinas em que o "hardware" comporta instruções especiais de empilhamento e desempilhamento de parâmetros, variáveis locais, endereço de retorno e "status" da máquina, a chamada de procedimentos implica em um certo "overhead". Além disso, algumas linguagens de programação (ou sua implementação em um dado sistema) podem não permitir chamadas recursivas de procedimentos. A eliminação de recursão foi uma das primeiras técnicas de transformações de programas que apareceram. Para o nosso caso, vamos aplicar um esquema descrito por Bauer et al. [2]:

(E12)

<u>função</u> f(<u>t</u>x)<u>t</u>':

  <u>se</u> B <u>então</u> ⌈S;f(x')⌋

    <u>senão</u> ⌈T;p(x)⌋



<u>função</u> f(<u>t</u>y)<u>t</u>':

  ⌈<u>t</u>x;

   x ← y;

   <u>enquanto</u> B <u>faça</u> ⟦S;x ← x'⟧

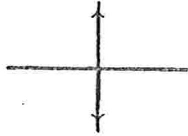
   T; p(x)⌋

onde x' é uma expressão (eventualmente em x).

Para aplicarmos E12, vamos inverter inicialmente a ordem das expressões selecionadas pelo comando <u>se</u>, usando o esquema derivado diretamente da definição desse comando:

(E13)

se E então E_1 senão E_2



se ⌉E então E_2 senão E_1

Obtemos

(14)

(15)

⌈função primo$_2$(int p',q') log:

se q'≠2 então p'mod(q'-1)≠0∧primo$_2$(p',q'-1)

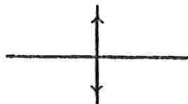
senão verdadeiro;

primo$_2$(p,p)⌋

Para aplicarmos E12, ainda falta isolar a chamada recursiva de primo$_2$. Para isso, empreguemos o seguinte esquema cuja demonstração pode ser feita trivialmente usando-se uma tabela de valores ("truth table"); E_i são expressões que assumem valores lógicos:

(E14)

se E então $E_1 \wedge E_2$ senão E_3



se $E \wedge E_1$ então E_2

senão se ⌉E então E_3 senão falso

Obtemos:

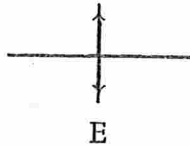
(15)

(16)

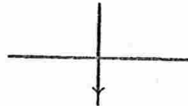
```
⌈função primo₂(int p',q') log:
        se q'≠2∧p'mod(q'-1)≠0
          então primo₂(p',q'-1)
          senão se q'=2 então verdadeiro senão falso;
 primo₂(p,p)⌋
```

mas

(E15) se E então verdadeiro senão falso

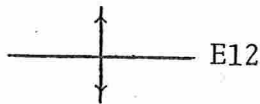


(16)



(17)

```
⌈função primo₂(int p',q') log:
        se q'≠2∧p'mod(q'-1)≠0 então primo₂(p',q'-1)
                               senão q'=2;
 primo₂(p,p)⌋
```



(18)

```
⌈função primo₂(p'',q'') log:
      ⌈⌈int p',q';
        (p',q') ← (p'',q'');
        enquanto q'≠2∧p'mod(q'-1)≠0
             faça (p',q') ← (p',q'-1);
        q'=2⌋⌋
 primo₂(p,p)⌋
```

Como p'' não muda de valor durante o processamento do procedimento e p' durante a iteração, e

$$(p',q') \leftarrow (p',q'-1) \equiv q' \leftarrow q'-1,$$

podemos escrever:

(18)

⌈ <u>função</u> $primo_2$(p'',q'') <u>log</u>:
  ‖⌈ <u>int</u> q';
    q' ← q'';
(19)   <u>enquanto</u> q'≠2∧p'<u>mod</u>(q'-1)≠0
      <u>faça</u> q' ← q'-1;
    q=2 ⌋‖
  $primo_2$(p,p)⌋

⌈ <u>int</u> q;
(21) q ← p;
<u>enquanto</u> q≠2∧p <u>mod</u> (q-1)≠0 <u>faça</u> q ← q-1;
q=2⌋

Esta será a versão final desda dedução. Lembremos que p é dado nas pré-condições C, e por isso não consta do bloco acima apresentado.

<u>Observações</u>:

1) Aparentemente, só há uma versão mais eficiente desse

algoritmo (dado o predicado inicial tal como ele foi formulado), que é a seguinte:

```
          ⌈int q;
(22')      q ← p-1;
           enquanto q≠1∧p mod q≠0 faça q ← q-1;
           q=1⌋
```

Teríamos chegado pelo mesmo caminho exatamente a essa versão se tivéssemos feito (as setas indicam as transformações feitas):

(4)

↓

```
        ⌈função primo (int p') log:
              ‖⌈função nprimo(int p'') log: nprimo2(p'',p'');
(4'a)           função nprimo2(int p''',q'''):
                      ∃ int q (q≤q'''-1∧q>1∧p'''mod q=0);
                ¬nprimo(p')⌋‖;
         primo(p)⌋
```

↓

```
        ⌈função primo(int p') log:
              ‖⌈função nprimo(int p'') log: nprimo2)p'',p''-1)
(4'b)           função nprimo2(int p''',q'''):
                      ∃ nit q (q≤q'''∧q>1∧p'''mod q=0);
                ¬nprimo(p')⌋‖;
         primo(p)⌋
```

A tautologia a ser introduzida em (5') seria agora $q''' = 1 \vee q''' \neq 1$, e assim por diante.

É talvez interessante comentar que ao autor nunca havia ocorrido que versões tão simples do algoritmo como (20) e (21) pudessem ser formuladas; mesmo em cursos em que esse exemplo foi dado aos alunos, versões mais complicadas, menos eficientes e, porque não, menos claras foram por ele desenvolvidas.

2) Como muitos programadores sabem, há algoritmos mais eficientes para se determinar se um número é primo ou não. Por exemplo, é suficiente testar-se apenas os ímpares, pulando-se os pares. Mas para isso era necessário dar um outro predicado inicial. Em lugar de

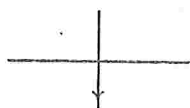
$$\neg\exists\ \underline{\text{int}}\ q:\ q<p \wedge q>1 \wedge p\ \underline{\text{mod}}=0$$

que, para um matemático, exprime tudo o que se deseja que um número primo satisfaça (e que encara o predicado como uma definição, sem preocupar-se com os aspectos computacionais), poderíamos formular o predicado da seguinte maneira:

$$p=2 \vee p\ \underline{\text{mod}}\ 2\neq 0 \wedge \neg\exists\ \underline{\text{int}}\ q(q<p \wedge q>1 \wedge q\ \underline{\text{mod}}\ 2\neq 0 \wedge p\ \underline{\text{mod}}\ q=0)$$

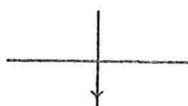
Numa dedução mais rápida, sem aplicarmos todos os passos formais, poderíamos ter (os números das versões correspondem aproximadamente às versões semelhantes na dedução anterior):

```
⌈ se p=2
    então verdadeiro
    senão se p mod 2=0
              então falso
              senão ¬ ∃ int q (q<p∧q>1∧q mod 2≠0∧
                                   p mod q=0)⌋
```



(4'')

```
⌈função nprimo2(int p',q') log:
        ∃ nit q((q'=3vq'≠3)∧q<q'∧q>1∧q mod 2≠0∧
                                      p mod q=0);
   se p=2 então verdadeiro
        senão se p mod 2=0 então falso
                         senão ¬ nprimo2(p',p')⌋
```

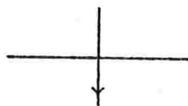


(8'')

```
⌈função nprimo2(int p',q') log:
        se q'=3 então falso
               senão ∃ int q: (q=q'-2vq≠q'-2)∧q<q'∧
                             q>1∧q mod 2≠0∧p'mod q=0;
   se p=2 então ... ⌋
```

(10'')

```
⌈função nprimo2(int p',q') log:
        se q'=3 então falso
               senão p'mod(q'-2)≠0v ∃ int q: q<q'-2∧q>1∧q mod 2≠0∧
                                                  p' mod q=0;
   se p=2 então ... ⌋
```



(12'')

```
⌈função nprimo2(int p',q') log:
        se q'=3 então falso
               senão p'mod(q'-2)≠0vnprimo2(p',q'-2);
   se p=2 então ... ⌋
```

```
(13'')  ⌈função primo₂(int p',q') log:
             se q'=3 então verdadeiro
                  senão p'mod(q'-2)=0∧primo₂(p',q'-2);
         se p=2 então verdadeiro
             senão se p mod 2=0 então falso
                           senão primo₂(p,p)⌋
```

```
(19'')  ⌈função primo₂(int p',q') log:
             ⟦nit q''
              q'' ← q';
              enquanto q''≠3∧p'mod(q''-2)≠0 faça q'' ← q''-2;
              q=3 ⟧
         se p=2 então ... ⌋
```

```
(20'')  ⌈se p=2 então verdadeiro
             senão se p mod 2=0
                       então falso
                       senão ⟦int q;
                              q ← p;
                              enquanto q≠3∧p mod(q-2)≠0
                                   faça q ← q-2;
                              q=3 ⟧
                                        ⌋
```

Finalmente, muitos programadores iriam reclamar que, para testar se um número é primo basta testar os ímpares que,

a partir de 5 se alternam uma vez com diferença de 2, em seguida de 4, novamente de 2, novamente de 4, etc., (o que pode ser facilmente provado), e que não é necessário testar-se apenas os números menores ou iguais a $\lceil\sqrt{p}\rceil$. Esta última condição é facilmente introduzida, tanto no predicado ($q<\lceil\sqrt{p}\rceil$) quanto no desenvolvimento; na versão final, por exemplo,(20), teríamos, em lugar de $q \leftarrow p$, $q \leftarrow$ inteiro(rzqd(real(p))), supondo a existência dessas 3 funções óbvias. Já para a primeira condição, duas opções se apresentam: modificar o predicado inicial e fazer toda a dedução novamente, ou alterar diretamente a versão final. Um novo predicado inicial poderia ser:

$$p=2 \vee p=3 \vee p \ \underline{\text{mod}}\ 2\neq 0 \wedge p \ \underline{\text{mod}}\ 3\neq 0 \wedge \neg\exists\ \underline{\text{int}}\ q\ (q<\lceil\sqrt{p}\rceil\ q>1 \wedge$$
$$\wedge q \ \underline{\text{mod}}\ 2\neq 0 \wedge q \ \underline{\text{mod}}\ 3\neq 0 \wedge p \ \underline{\text{mod}}\ q=0)$$

Uma modificação na versão final não é difícil de ser feita, mas neste caso fugir-se-ia do método e não se teria certeza da "corretude" do programa obtido.

3) Note-se que, na prática, a dedução pode ser muito mais rápida, como demonstramos na observação anterior. O programador experimentado pode rapidamente aplicar um ou vários esquemas sem desenvolver todos os detalhes, e sem com isso prejudicar a "corretude" do programa que ele deduzirá. Além disso, é interessante observar que a ingeniosidade do programador é aplicada neste método na escolha dos esquemas a serem utilizados, a determinação da frequência de sua utilização e, eventualmente, na dedução e prova de novos esquemas. Por outro lado, a descrição do problema e dos dados usando todo o ferramental do cálculo de predicados deve facilitar a "corretude" dessa descrição. Compare-se a descrição compacta do predicado inicial com o programa correspondente deduzido

(20") para que se conclua pelo interesse do uso do método. Além disso, cremos que dessa maneira consegue-se um ponto em comum entre o programador e o matemático; o primeiro pode usar diretamente a linguagem desse último para o desenvolvimento dos programas requeridos por este.

4) É interessante examinar com certa atenção algumas diferenças entre as versões (14) e (20), isto é, entre as formas recursiva e iterativa. A forma recursiva é, aparentemente, mais clara, de compreensão mais imediata. Essa distinção ficaria ainda mais patente se em lugar de (14) fizéssemos a versão (14') resultante das transformações intermediárias ($4'_a$) e ($4'_b$):

(14')

⌈ <u>função</u> $primo_2$(<u>int</u> p',q') <u>log</u>:
  <u>se</u> q'=1 <u>então</u> <u>verdadeiro</u>
    <u>senão</u> p'<u>mod</u> q'≠0∧$primo_2$(p',q'-1);
  $primo_2$(p,p-1) ⌋

Na verdade, essa rotina está testando a divisibilidade de p por todos os inteiros maiores do que a unidade; ela dá o valor (verdadeiro ou falso) do seguinte predicado: "p não é divisível por nenhum de seus predecessores inteiros q maiores do que 1". Ou, indutivamente, "se q = 1 então verdadeiro, senão se verdadeiro para (isto é, não é divisível por) q, teste o predicado para q-1".

Em segundo lugar, note-se que nas versões recursivas não há variáveis locais, somente parâmetros. No entender de Bauer [1], a presença de variáveis indica níveis baixos no conceito de programação "top-down". O comando de atribuição também só apareceu nas versões de nível conceitual "inferior". O método descrito mostra que o nível formal é

também "inferior". Nesse sentido, a programação usando-se sentenças do cálculo de predicados é a de mais alto nível formal possível (acima desta, talvez somente a linguagem natural, mas aí escapamos do formalismo). No método descrito, os níveis "inferiores" foram deduzidos a partir dos "superiores", não sendo portanto "prejudiciais" ou indicadores de "maus hábitos de programação"; não houve, propriamente uma programação em "baixo nível", e sim dedução de um programa nesse nível. Partsch e Pepper [24] mostram como as transformações podem ser conduzidas até a geração de programas em linguagem de máquina o que, aliás, os compiladores já o fazem sob certo aspecto. Em meio às deduções, pode-se dar eventualmente o aparecimento do mal-afamado "goto" [13], sem que com isso tenha se incorrido em "má programação".

5) Bauer et al.[3] dão um exemplo de dedução de um algoritmo para determinar se um número é ou não primo. Ele difere bastante do nosso, que foi desenvolvido independentemente, nos seguintes pontos: i) O predicado inicial usa um quantificador universal. ii) Em lugar do operador mod — que em nosso exemplo não é desenvolvido — é usado inicialmente um predicado "divide", para o qual também é desenvolvido um algoritmo. iii) A ênfase do desenvolvimento é colocada em "divide" e, posteriormente, em uma outra versão, em mod. iv) Conforme chamam a atenção os próprios autores (pg.42), não foi feita uma dedução estritamente formal; assim, contrariamente ao nosso exemplo, não é apresentada uma prova de preservação da "corretude" de cada versão apresentada.

- ¤ -

CAPÍTULO 3

CONSULTAS A BASES DE DADOS RELACIONAIS FORMULADAS EM CÁLCULO DE PREDICADOS

3.1 - INTRODUÇÃO

Neste capítulo faremos uma breve introdução ao Modelo Relacional de Dados, e desenvolveremos uma linguagem de consultas ("CONSULTOL") baseada em cálculo de predicados, para possibilitar a dedução de algoritmos de consultas nos capítulos seguintes.

O Modelo Relacional foi introduzido por Cood [8]. Uma excelente exposição sobre esse modelo, pode ser encontrada em [20]. Nesse modelo, dados são agrupados em conjuntos de registros. Cada elemento de um conjunto R de registros é uma n-pla ordenada, sendo que cada elemento de ordem i (i=1,2,...,n) de uma n-pla pertence a um conjunto de dados simples D_i. Assim, temos o análogo a uma Relação R no sentido matemático:

$$R \subseteq D_1 \times D_2 \times \ldots \times D_n \quad \text{(Produto Cartesiano)}$$

isto é, cada registro é um elemento de R; D_i é denominado domínio da relação R, e constitui-se em um conjunto de dados,

em geral tendo a mesma representação. Para simplificar, usamos a notação $R \equiv (D_1, D_2, \ldots, D_n)$.

Note-se que essa caracterização de dados está de acordo com a utilização tradicional de arquivos em computadores; com raríssimas exceções, os arquivos tradicionais são em geral conjuntos de registros em que cada registro contém dados seguindo, dentro dos registros, uma ordem fixa segundo sua característica. Representando-se um arquivo como uma relação matemática, pode-se utilizar todo o formalismo da teoria das relações no que for pertinente ao problema; em particular, todo o ferramental da teoria dos conjuntos torna-se accessível, tanto para a descrição como para a manipulação dos arquivos. Uma relação pode ser representada por uma tabela, maneira tradicional de se apresentar registros de um arquivo.

É interessante notar-se que o próprio Codd [9] chamou a atenção para o fato de que o produto cartesiano da Álgebra difere daquilo que se costuma denominar pelo mesmo termo termo, na literatura de Bases de Dados. De fato, dadas duas relações R e S com, respectivamente, n-plas de r e s elementos, na Álgebra $R \times S$ é um conjunto de duplas. Em cada dupla, o 1º elemento é uma r-pla e o 2º elemento é uma s-pla. Já em Bases de Dados $R \times S$ indica um conjunto de n-plas com $r + s$ elementos cada uma. Daí o fato de Codd ter usado um sinal especial para o produto cartesiano da Base de Dados, qual seja o $\otimes$. Ele define esse produto em termos de concatenação: se $r \cdot s$ é a concatenação da n-pla r com a n-pla s,

$$R \otimes S = \{(r \cdot s) \mid r \in R \wedge s \in S\}.$$

Entendida a ressalva sobre o produto cartesiano, usaremos o símbolo usual $\times$ para indicar o $\otimes$ de Codd.

Por exemplo, podemos representar um arquivo contendo algumas informações sobre funcionários de uma empresa através da seguinte relação:

$R \subseteq$ nome-do-funcionário × salário × nome-do-gerente onde "nome-do-funcionário", "salário" e "nome-do-gerente" são nomes dos conjuntos de dados que constituem os domínios de R. Se

nome-do-funcionário = {IVAN, WALDYR, BETE, INES, GRAÇA, CYRO, REGINA, ORLANDO, PAULO, ERNESTO}

salário = {10000, 15000, 20000, 25000, 30000, 35000}

nome do gerente = nome-do-funcionário

poderíamos representar uma relação R descrevendo algumas informações sobre os funcionários de uma empresa hipotética por:

$R \equiv$ (nome-do-funcionário, salário, nome-do-gerente)

REGINA	10000	CYRO
BETE	15000	CYRO
INES	15000	CYRO
GRAÇA	15000	CYRO
CYRO	20000	WALDYR
WALDYR	30000	ORLANDO
ORLANDO	35000	PAULO
PAULO	30000	ERNESTO
ERNESTO	40000	ERNESTO

Uma representação rigorosa seria

R = {(REGINA,10000,CYRO), (BETE,15000,CYRO),...}

Codd introduziu [9] dois tipos de linguagens formalizadas de consultas a bancos de dados relacionais, isto é,

aqueles que se constituem de conjuntos de relações; ele chamou essas linguagens de "Álgebra das Relações" e "Cálculo das Relações". A primeira constitui-se em uma linguagem de expressões contendo relações e operadores sobre relações e que é, em síntese, uma extensão das expressões da álgebra dos conjuntos. A segunda, particularizada em uma linguagem por ele denominada de ALPHA, é um conjunto de sentenças do cálculo de predicados de primeira ordem formadas segundo certas regras; as variáveis das sentenças assumem os valores de n-plas das relações, podendo ser projetadas em um ou mais de seus domínios. Neste trabalho, estaremos interessados somente no "Cálculo de Relações"; Codd mostrou [9] que a álgebra de relações conforme definida em [9] é "relacionalmente completa", isto é, qualquer consulta formulada em ALPHA pode ser convertida numa consulta da linguagem da álgebra de relações citada. Uma boa referência a esse respeito é Lapyda [20].

No restante do trabalho, suporemos que as relações estejam em "Primeira Forma Normal" (Codd [10]), isto é, todas as n-plas têm todos os seus elementos definidos e os domínios são simples, isto é, não são formados por sua vez por outros domínios.

3.2 - EXEMPLOS DE CONSULTAS EM ALPHA

Para exemplificar brevemente a linguagem ALPHA, damos abaixo algumas consultas expressas em linguagem natural e naquela linguagem, ligeiramente modificada por nós para maior legibilidade. Uma grande quantidade de exemplos, com a notação original de Codd, pode ser encontrada em Lapyda [20]). Usaremos a relação R anteriormente introduzida. Os exemplos são suficientemente simples para que a sintaxe seja auto-ex

plicativa.

Suponhamos que as variáveis r, r' e r" assumam o valor das n-plas de R (isto é, percorram R):

C_1: "Dê o nome dos funcionários de R"
consulta em ALPHA: (r. nome-do-funcionário)

C_2: "Dê o nome dos funcionários que são gerentes":
(r. nome-do-gerente)

C_3: "Dê o nome dos funcionários que ganham 20000 ou mais, bem como seu respectivo salário e gerente".
(r): r. salário ≥ 20000

C_4: "Dê o nome e salário dos funcionários que são gerentes":
(r.nome-do-funcionário, r.salário): ∃ r'(r.nome-do-funcionário= r'.nome-do-gerente)

C_5: "Dê o nome dos funcionários que não são gerentes":
(r.nome-do-funcionário): ¬∃ r'(r'.nome-do-gerente = r.nome-do-funcionário)

C_6: "Dê o nome dos funcionários cujos gerentes ganham 30000":
(r.nome-do-funcionário): ∃ r'(r'.nome-do-funcionário = r.nome-do-gerente∧r'.salário = 30000)

C_7: "Dê o nome dos funcionários cujos gerentes somente chefiam funcionários que ganham mais do que 15000":
(r.nome-do-funcionário): ∀r'(r.nome-do-gerente ≠ r'.nome-do-gerente∨r'.salário > 15000)

C_8: "Dê o nome dos funcionários que ganham mais do que seus gerentes":
(r.nome-do-funcionário): ∃ r'(r.nome-do-gerente = r'.nome-do-funcionário∧r.salário>r'.salário)

C_9: "Dê o nome dos funcionários que ganham mais do que seus gerentes, no caso em que estes só chefiam funcionários que ganham

mais do que 15000"

(r.nome-do-funcionário): $\exists$ r'(r.nome-do-gerente = r'.nome-do-funcionário $\wedge$ r.salário>r'.salário $\wedge$ $\forall$r"(r.nome-do-gerente $\neq$ r".nome-do-gerente $\vee$ r".salário > 15000))

As seguintes observações são pertinentes:

1) O resultado da consulta é sempre uma relação.

2) O sinal ":" pode ser lido como "tal que".

3) As variáveis que aparecem à esquerda de ":" , denominadas de variáveis-objetivo, são as únicas variáveis livres na sentença do lado direito de ":" , denominado de qualificação; outras variáveis no lado direito de ":" devem ser ligadas por quantificadores.

4) Para se compreender "fisicamente" como pode-se processar uma consulta, basta supor que as variáveis-objetivo assumem sucessivamente, como seu valor, todas as n-plas da relação. Para cada n-pla, verifica-se se a expressão lógica que constitui a qualificação é verdadeira, caso em que se inclui na relação-resultado as projeções desejadas das variáveis-objetivo para essa n-pla. Se a expressão lógica for falsa nada se faz. Em ambos os casos, as variáveis livres assumem o valor de uma outra n-pla (eventualmente a próxima), testa-se novamente a qualificação, etc.

5) Se na qualificação aparecer um quantificador existencial, "varre-se" a relação à qual a variável assim ligada está associada, mantendo-se fixos os "valores" (isto é, as n-plas) "apontadas" pelas outras variáveis da sentença que é o escopo do quantificador. Se para algum valor da variável ligada a sentença do escopo for verdadeira, a expressão com o quantificador será verdadeira; caso contrário, será falsa.

Para o quantificador universal, mantendo-se todas as variáveis livres do escopo com o valor que elas tem, a expressão englobada pelo quantificador será verdadeira se, "varrendo-se" a relação associada à variável ligada, todos os seus valores derem como resultado um valor verdadeiro para essa expressão.

6) Em nosso exemplo usamos, para maior simplicidade, apenas uma relação. Nada impede que sejam usadas várias relações; de fato, nas qualificações em que aparece mais de uma variável, tudo se passa como se cada variável "apontasse" para uma relação independente.

A linguagem ALPHA de Codd é relativamente limitada. Por exemplo, ele só usa os conectivos básicos $\neg$, $\wedge$ e $\vee$. Na consulta C_7, o mais natural seria uma formulação em que aparecesse a implicação lógica:

C_7: (r.nome-do-funcionário): $\forall$r' (r.nome-do-gerente = r'.nome-do-gerente $\Longrightarrow$ r'.salário > 15K)

que evidentemente é equivalente à anterior, pois $a \Rightarrow b \equiv \neg a \vee b$ a mesma observação aplica-se para C_9.

3.3 - A LINGUAGEM CONSULTOL

3.3.1 - Introdução

Passemos a estender a linguagem ALPHA, no sentido de:

1) Aproximá-la mais das linguagens de programação, principalmente no tocante à declaração de dados e suas estruturas.

2) Aproximá-la da notação matemática usual para conjun-

tos, já que sendo as relações conjuntos de n-plas, o que se está fazendo ao formular uma consulta em cálculo de predicados é definir um conjunto de n-plas que constituirá o resultado esperado.

3) Permitir todo o ferramental da notação usual de conjuntos e de funções. Por exemplo, facilitando a formulação de consultas como "dê o nome de todos os funcionários cujos salários sejam superiores à média dos salários de todos os funcionários".

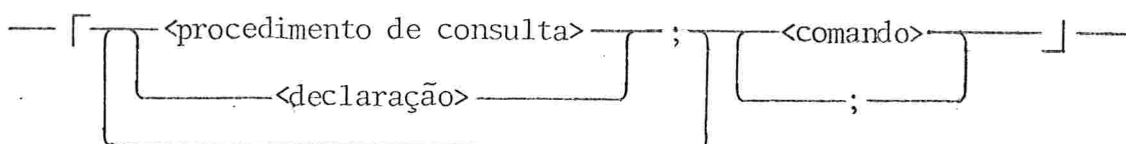
É interessante notar que várias linguagens foram definidas a fim de se fazer consultas a bancos de dados, independentemente de se ter que constuir um programa usando as linguagens tradicionais (que se tornam assim as chamadas "linguagens hospedeiras") como base. Essas "linguagens de consulta" ("Query Languages") são em geral versões da ALPHA de Codd, muitas vezes dissimulando-se os quantificadores,"abaixando-se" assim o seu nível. É o caso, por exemplo, das linguagens QUEL [18] e SEQUEL [7]. Nossa intenção com a introdução do CONSULTOL não é a de criar mais uma linguagem de programação, e sim permitir a formulação de consultas usando uma notação praticamente matemática. No capítulo seguinte introduziremos esquemas de transformação que permitem, a partir de consultas formuladas em CONSULTOL, sintetizar programas dentro das linhas traçadas nos capítulos 1 e 2.

A linguagem CONSULTOL para declaração de relações e formulação de Consultas será apresentada através de uma gramática ambígua dada pelo diagrama abaixo (a razão de termos definido uma gramática ambígua é de que desta maneira ela fica mais sucinta); a semântica informal dada em continuação elimina as ambiguidades e dá especificações daquilo que não é óbvio ou comum à maioria das linguagens de programação.

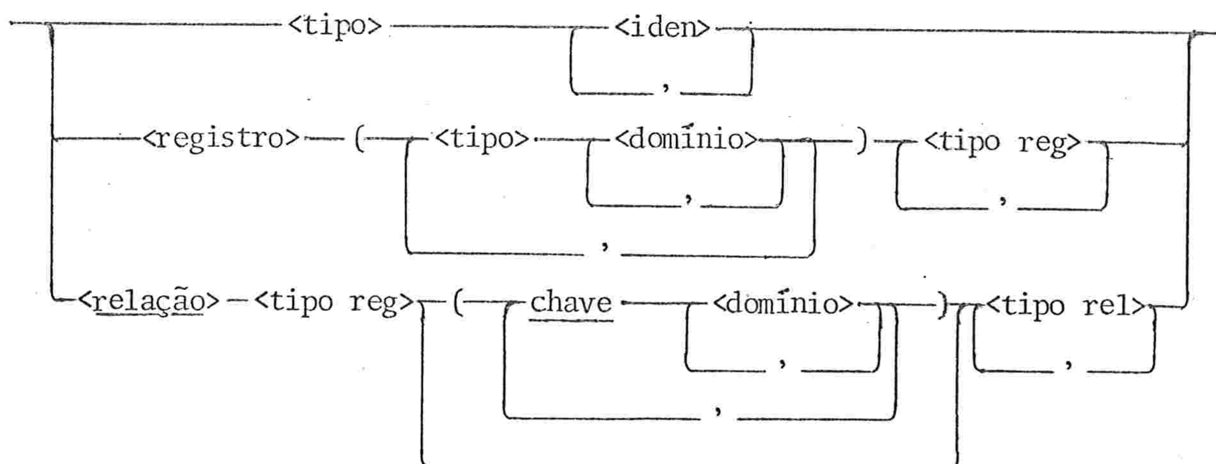
O diagrama sintático que usamos é derivado dos empregados por N. Wirth na descrição da linguagem PASCAL [27] e cuja origem cremos ser devida a Conway [11]. Para simplificar, usaremos a notação BNF [23] onde ela é mais concisa do que o correspondente diagrama. Não-terminais são colocados entre colchetes angulares ("<" e ">") e terminais são grifados, quando não são símbolos especiais.

3.3.2 - Sintaxe da linguagem CONSULTOL

<programa>

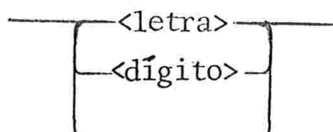


<declaração>



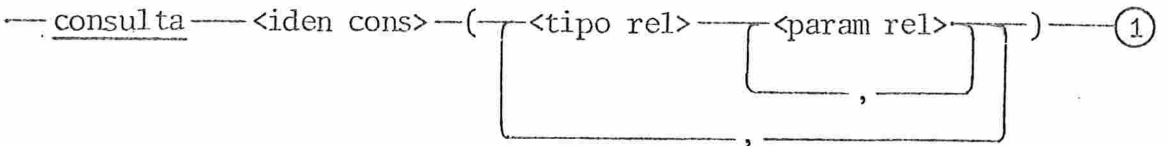
<tipo>::= *int* | *real* | *log* | *cadeia* | <tipo reg> | <tipo rel>

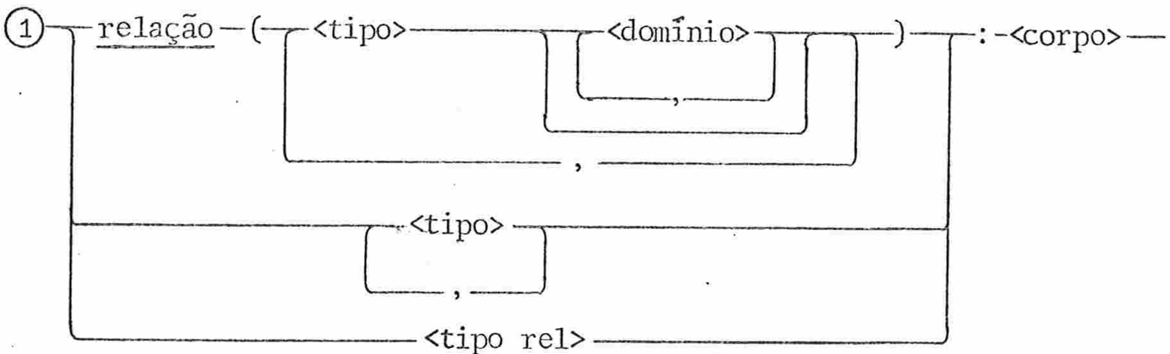
<iden>



<letra>::= A | B | ... | Z
<digito>::= 0 | 1 | ... | 9
<tipo reg>::= <iden>
<tipo rel>::= <iden>
<domínio>::= <iden>

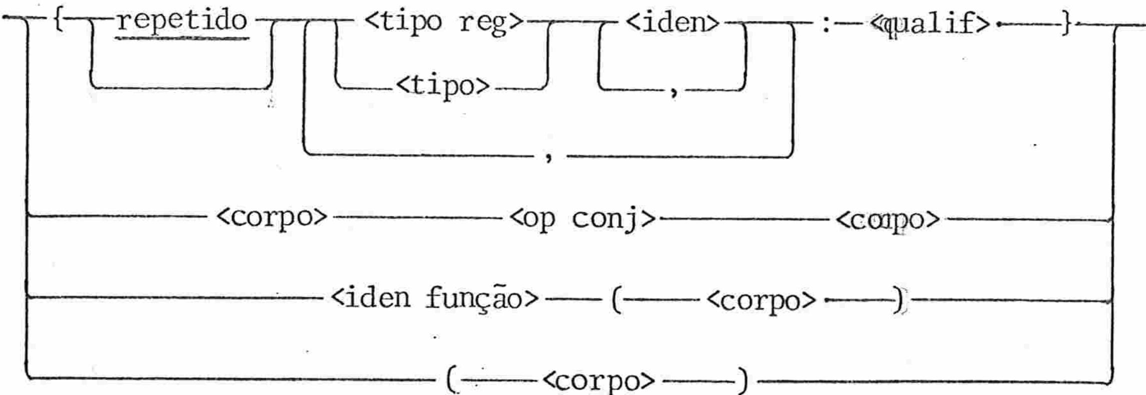
<procedimento de consulta>





<iden cons>::= <iden>
<param rel>::= <iden>

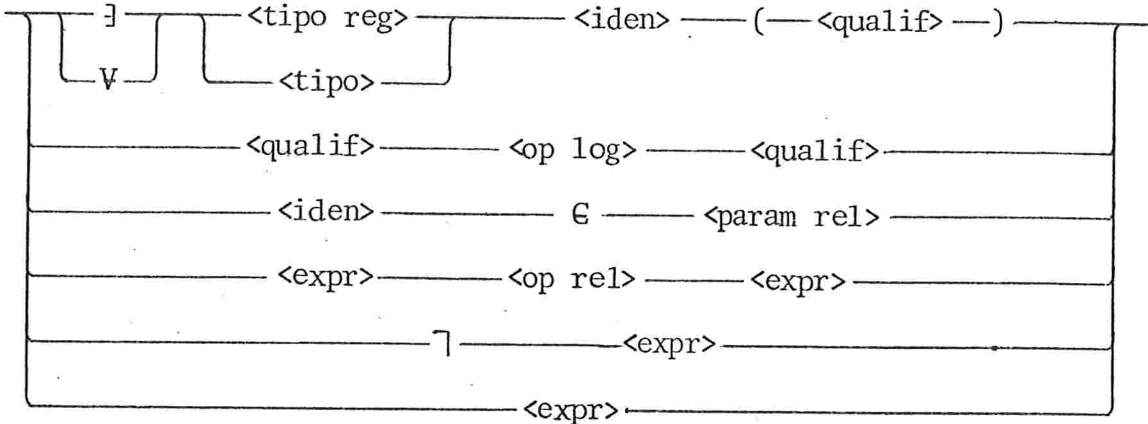
<corpo>



<op conj>::= ∪ | ∩ | -

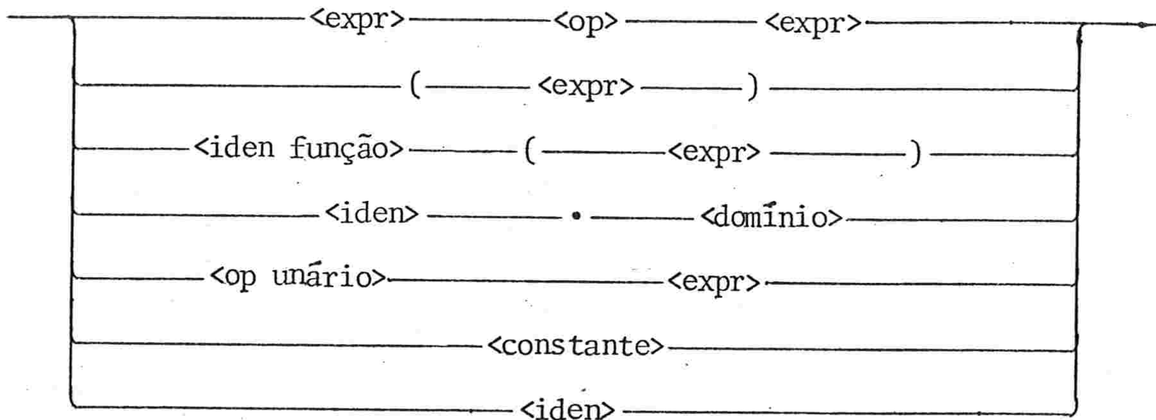
<iden função>::= <iden>

<qualif>



<op rel>::= < | > | = | ≠ | ≥ | ≤ <op log>: = ∧|∨|<u>orex</u>| ≡ | ⇒

<expr>



<op>::= + | - | * | / | ** <op unário>::=+|-

<constante>::= constante numérica | "<cadeia>"

<cadeia>::= cadeia de caracteres

<comando>::= comandos usuais de uma linguagem tipo ALGOL

Comentários podem ser inseridos em qualquer parte do programa, e são delimitados por "<<" e ">>"

3.3.3 - Descrição informal da semântica da linguagem CONSULTOL

No que segue, descreveremos sucintamente a semântica de cada não-terminal mais importante. Não cabe aqui uma descrição completa e nem formal; os exemplos dados mais adiante servem para ilustrar parcialmente as construções mais típicas. Não discorreremos sobre aquilo que cremos ser óbvio, seja devido aos nomes de terminais e não-terminais, que já de per si sugerem seu significado, seja pela semelhança com linguagens de programação do tipo ALGOL-60 [23] ou da manipulação comum de bases de dados relacionais.

a. <programa>. Os sinais ⌜ e ⌟, que correspondem aos tradicionais begin ... end podem ser usados em diversas variantes, para maior clareza: ║⌜... ⌟║, ╒ ... ╛, ⌜~ ... ~⌟ etc.

b. <declaração>. Além da declaração de tipos simples, introduzimos a declaração de registros e de relações; estas são conjuntos de n-plas, todas com a mesma estrutura de um determinado registro; as chaves podem ser domínios simples (isto é, cada valor de um desses domínios determina univocamente uma n-pla da relação), ou domínios compostos (isto é, somente a sub-n-upla da chave é que determina univocamente uma n-pla da relação). Ex: (chave A,B, chave C, chave D) indica que (A,B), C e D são as três chaves da relação.

c. <tipo>. log é abreviação de lógico, isto é, booleano. cadeia é um tipo que aqui não especificamos com seu comprimento, mas que em implementações deveria certamente contê-lo; por exemplo, cadeia 20 A, B indicaria que A e B são cadeias de 20 caracteres.

d. <procedimento de consulta>. Esses procedimentos são análogos aos procedimentos de linguagem tipo ALGOL-68 [26], tendo um nome (<iden cons>), parâmetros somente do tipo rela

ção e retornando valores que podem ser relações (em que são especificados os nomes dos domínios) ou tipos simples. Se o resultado da consulta precisar ser armazenado, deve-se fazer uma atribuição da chamada da consulta (que se processa da maneira habitual de chamadas de procedimentos tipo função) a uma variável declarada convenientemente. Note-se que fizemos uma extensão ao ALGOL-68: o resultado da consulta pode ser uma n-pla, colocando-se antes de:<corpo> uma sequência de tipo simples, por exemplo: <u>int</u>, <u>cadeia</u>, <u>cadeia</u>.

e.<corpo>. A ambiguidade sintática desse não-terminal é resolvida dando-se a seguinte precedência, em ordem decrescente, a <op conj>: ∩, ∪, − (que é a complementação relativa de conjuntos). <u>repetido</u> significa que o "conjunto" resultante tem elementos repetidos. Se a declaração das variáveis-objetivo (cf. observação (3) anterior) contiver um <tipo reg>, isto é um tipo registro, os nomes dos domínios associados a esse tipo ficarão associados à n-pla resultante, e portanto o resultado da consulta também receberá esses nomes de domínios. Nesse caso, em <procedimento de consulta> deve-se tomar o caminho que contorna <domínio>, deixando este vazio.

f.<qualif>. A ambiguidade das qualificações é resolvida pela seguinte precedência decrescente de <op log>:∧,∨ e <u>orex</u>, ≡,⟹. Nas qualificações com quantificadores, <iden> é a variável ligada que deverá aparecer dentro da <qualif> que a segue entre parênteses. Note-se que com essa sintaxe não permitimos algo como ∃ <u>int</u> x ∃ <u>log</u> y ..., que deverá ser escrito ∃ <u>int</u> x (∃ <u>log</u> y ...)). O uso dos quantificadores serve para produzirmos um relacionamento entre uma n-pla e uma ou mais n-plas da mesma ou de outra relação. Quando a qualificação refere-se apenas a relacionamentos entre elementos de cada n-pla, não é necessário usar quantificadores.

g. <expr>. A ambiguidade é resolvida pela seguinte precedência decrescente de <op>: **, / e *, + e -. A construção <iden>•<domínio> permite a referência a um elemento de um registro; neste caso <iden> deve ser o tipo registro.

h. <comando>. Deixamos propositadamente de definir os comandos da linguagem; todos os comandos mais comuns das linguagens algorítmicas deverão ser aceitos, com a ressalva de que blocos, comandos iterativos e seletivos devem ser também expressões, isto é, retornam um valor, como em ALGOL-68 [26].Entrada e saída de relações também devem estar presentes.

3.4 - EXEMPLOS DE CONSULTAS EM CONSULTOL

Formulamos, a seguir, as consultas C_1 a C_9 em CONSULTOL; por meio de comentários explicaremos o significado de cada uma:

⌈ <<declaração da estrutura da relação, isto é, de cada n-pla; daí para frente o nome do registro funciona como tipo e será grifado>>;

registro (cadeia nome-do-funcionário, int salário, cadeia nome-do-gerente) FUNCIONÁRIO;

<<declaração de tipo relação, isto é, conjunto de n-plas com a estrutura acima>>;

relação FUNCIONÁRIO (chave nome-do-funcionário) EMPRESA;

<<isto é, EMPRESA identifica uma relação em que cada n-pla tem a estrutura do tipo FUNCIONÁRIO; EMPRESA passa a ser um tipo e será grifada daqui em diante>>;

<<para efetuar-se a consulta C_1, declara-se uma consulta>>;

‖⌈ consulta C_1 (EMPRESA EMP) relação (cadeia NOME):
{cadeia y: ∃ FUNCIONÁRIO func (func ∈ EMP ∧ y= func.nome-do-funcionário)};

<<isto é, ao chamar-se essa consulta (como se faz com um procedimento), obtem-se como resultado uma relação com um só domínio (NOME); o argumento da consulta deve ser uma relação do tipo EMPRESA, isto é, um conjunto de n-plas de tipo FUNCIONÁRIO; note-se que no corpo da consulta calcula-se um conjunto de elementos tipo cadeia>>;

<<declaração de uma relação E do tipo EMPRESA>>;

EMPRESA E;

leia E;

imprima C_1(E);

<<dessa maneira, chamamos C_1 com argumento E; o resultado de C_1, isto é, uma relação com um único domínio (NOME), é impresso>> ⫼;

<<como estamos interessados unicamente nos procedimentos tipo consulta, isto é, na sua declaração, não faremos mais a declaração nem a leitura das relações que serão usadas como argumento, e nem a impressão (ou outra manipulação) dos resultados>>;

<<C_2 será pulada por ser semelhante a C_1>>;

consulta C_3 (EMPRESA EMP) relação FUNCIONÁRIO:

{FUNCIONÁRIO f: $f \in EMP \wedge f \cdot salário \geq 20000$};

<<note-se que neste caso não foi necessário introduzir na qualificação um quantificador, pois não há função (em particular, projeção) que é aplicada a cada n-pla para gerar um elemento do conjunto-resultado>>;

consulta C_4 (EMPRESA E) relação (cadeia N, int SAL):

{(cadeia N, int SAL): ∃ FUNCIONÁRIO G(G∈E∧N = G.nome-do-funcionário∧SAL = G.salário∧∃FUNCIONÁRIO F(F∈E∧F.nome-do-gerente = G.nome-do-funcionário))}

<<C_5 é semelhante a C_4 a menos da negação ¬ ∃FUNCIONÁRIO... e portanto será pulada>>;

<<C_6 também não traz nada de novo>>;

consulta C_7 (EMPRESA X) relação (cadeia FUNC):

{(cadeia FUNC): ∃FUNCIONÁRIO R(R∈X∧FUNC = R • nome-do-funcionário∧∀FUNCIONÁRIO R'(R'∈X⟹ R• nome-do-gerente≠R' •nome-do-gerente∨R• salário>15000)))};

<<ou>>

consulta C'_7 (EMPRESA X) relação (cadeia NOME):

{(cadeia NOME): ∃FUNCIONÁRIO R(R∈X∧NOME = R • nome-do-funcionário∧∀FUNCIONÁRIO R'(R'∈X⟹ NOME = R• nome-do-funcionário∧R • nome-do-gerente = R'•nome-do-gerente ⟹R'•salário >15000))};

<<note-se em ambos os casos acima, que o quantificador ∀ exige R'∈X ⟹ ... pois, a rigor, R' pode ser qualquer n-pla com estrutura FUNCIONÁRIO; nesse sentido, uma formulação R'∈X∧... produziria sempre o resultado falso para o termo que contém o quantificador ∀, pois há alguns R' que não pertencem a X...>>;

consulta C_8 (EMPRESA E) relação (cadeia NOME):

{cadeia x: ∃FUNCIONÁRIO F(F∈E∧x = F•nome-do-funcionário∧∃FUNCIONÁRIO G(G∈E∧F•nome-do-gerente = G•nome-do-funcionário∧F•salário>G•salário))};

consulta C_9 (EMPRESA E) relação (cadeia NOME):

{cadeia x: ∃FUNCIONÁRIO F(F∈E∧x = F•nome-do-funcioná
rio∧∃FUNCIONÁRIO G(G∈E∧F•nome-do-gerente = G
•nome-do-funcionário∧∀FUNCIONÁRIO F'(F'∈E∧F
•nome-do-gerente = F'•nome-do-gerente ⟹ F'•
salário>15000)))};

<<observações

1. Note-se que, em relação às consultas formuladas em ALPHA, as consultas em CONSULTOL têm um quantificador existencial a mais. Isso se deve ao fato de termos buscado uma formulação mais próxima possível da notação matemática usual; de fato, encontram-se frequentemente nos textos matemáticos definições de conjuntos como

$$\left\{x \in N: P(x)\right\}$$

onde P(x) é um predicado.

Se tivéssemos eliminado o primeiro quantificador existencial de cada consulta, teríamos que colocar uma função (em nosso caso, uma projeção), no lugar da variável livre cujos valores constituem o conjunto assim definido; teríamos algo como

$$\left\{f(x): P(x)\right\}$$

que, além de inortodoxo, tiraria uma padronização das consultas necessárias para as transformações a serem introduzidas no próximo capítulo. Como a consulta C_3 requer como resultado a n-pla completa, não havendo projeção pudemos formulá-la sem o quantificador existencial.

Lembremos que cadeia x equivale a "x pertence ao conjunto de cadeias".

2. Se mais de uma relação fosse usada em uma consulta, teria sido necessário usar mais de um argumento na declaração da consulta correspondente. Para exemplificar esse caso, suponhamos uma relação envolvendo projetos e os já conhecidos funcionários; lembrando que ainda estamos dentro do programa acima (e na área de declarações do bloco mais externo!), temos:>>

registro (cadeia nome-do-projeto, nome-do-funcionário) PROJFUNC;

relação PROJFUNC (chave nome-do-projeto, nome-do-funcionário) RPF;

<<façamos uma rotina de consulta para sabermos quais os gerentes dos funcionários que trabalham no projeto "PONTE":>>

consulta C_{10} (EMPRESA E, RPF P) relação (cadeia NOME):

{cadeia x: ∃FUNCIONÁRIO F(F∈E∧x = F•nome-do-gerente∧∃PROJFUNC PR(PR∈P∧PR•nome-do-projeto = "PONTE"∧F•nome-do-funcionário=PR•nome-do-funcionário))};

<<continuando as observações:

3. Para exemplificarmos o uso do ferramental matemático dentro da linguagem CONSULTOL e de um problema no seu uso, suponhamos que se deseje uma rotina para a seguinte consulta: "forneça os nomes dos funcionários de uma relação do tipo EMPRESA tais que seu salário seja superior à média dos salários de todos os funcionários da relação". Suponhamos que a função $MEDIA_i(X)$ calcule a média (inteira) dos números inteiros que constituem o i-ésimo elemento das n-plas de uma relação X:>>

consulta C_{11} (EMPRESA E) relação (cadeia NOME):

{cadeia N: ∃FUNCIONÁRIO F(F∈E∧N = F•nome-do-funcio-

rio∧F•salário>$MÉDIA_2$(E)};

<<4. Note-se na sintaxe do CONSULTOL que, precedendo a lista de variáveis livres, podemos ter a palavra reservada repetido; com isso podemos construir "conjuntos" com elementos repetidos; isso é essencial para podermos aplicar algumas funções que exigem a repetição para terem significado correto. Assim, seja construir uma rotina para a seguinte consulta: "dê a média dos salários dos funcionários chefiadas por CYRO:>>

consulta C_{12} (EMPRESA E) int:

MÉDIA({repetido int S: ∃FUNCIONÁRIO F(F∈E∧S=F•sa

lário∧F•nome-do-gerente = "CYRO"});

<<onde, em notação de funções, MÉDIA:{int} ⟶ int. Note-se que a construção do conjunto deve ser entendida segundo o esquema mencionado na observação (4) acima. Observe-se ainda que, na observação (3) não foi necessário usar o recurso da especificação da repetição pois um dos elementos das n-plas de relações do tipo EMPRESA é uma chave (isto é, não aparece repetido) segundo a declaração que encabeça a sequência de consultas em CONSULTOL.>> ⌋

- ¤ -

CAPÍTULO 4

ESQUEMAS PARA DEDUÇÃO DE PROGRAMAS DE CONSULTAS A BASES DE DADOS

4.1 - INTRODUÇÃO

Neste capítulo introduziremos as várias formas de consultas permitidas na linguagem CONSULTOL definida no capítulo anterior. A seguir, introduziremos esquemas de transformações dessas consultas em algoritmos "semi-executáveis", conforme a caracterização que fizemos à pág. 6. No capítulo seguinte faremos a aplicação desses esquemas aos exemplos de consultas vistos no capítulo anterior, e mostraremos como transformar os algoritmos "semi-executáveis" em "executáveis" eliminando o operador de escolha η.

4.2 - CONSULTAS SEM QUANTIFICADORES

Examinando <procedimento de consulta> na sintaxe da linguagem CONSULTOL (capítulo anterior), vemos que uma das rotinas de consulta mais simples tem a forma

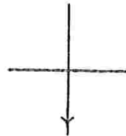
(F1) $\underline{\text{consulta}}\ C(\underline{R}X)\underline{R}:\{\underline{r}y : y \in X \wedge P(y)\}$

onde $\underline{R}$ é um tipo relação, relação essa que é um conjunto de

de registros tipo r, e P um predicado. Essa relação em síntese seleciona algumas n-plas de X baseado em alguma propriedade que, para cada n-pla, depende apenas desta. A consulta C_3 do capítulo anterior é um exemplo dessa forma de consulta.

O esquema que aplicaremos na sintetisação de um programa que realiza essa consulta é o seguinte (continuamos a numeração dos esquemas como interrompida no capítulo 2):

(E16) consulta C(RX)R:{ry:y∈X∧P(y)}



função C(RX)R:

se X=∅ então ∅

senão ⌈ηry:y∈X;

(se P(y) então {y} senão ∅)∪C(X-{y})⌋

onde ∅ é o conjunto vazio.

Note-se que no procedimento função C deixamos a região das consultas a bancos de dados entrando-se na nomenclatura das linguagens de programação. O tipo "conjunto" não existe nas linguagens comuns de programação, ocorrendo apenas em algumas experimentais, como por exemplo SETL [25]; no entanto, não é difícil de ser implementado, por exemplo através de um arquivo sequencial, em que na introdução de um novo elemento busca-se um outro igual já existente na sequência; se esta ocorrer, não se introduz o novo elemento. Se a sequência for ordenada (segundo a chave da relação), pode-se empregar um processo mais eficiente de busca, por exemplo a busca binária; no entanto para este tipo de sequência

a inserção implica em deslocamento de vários registros. Uma outra possibilidade é o uso de cálculo de endereços ("hashing"), que é em geral o método de busca mais rápido e que não tem grandes problemas de inserção. Com essa digressão quisemos apenas indicar que o uso de um tipo abstrato como "conjunto" pode ser implementado na prática. Por outro lado, o uso desse tipo permite que, posteriormente, se particularize a estrutura dos arquivos sem perda da "corretude" das deduções, obtendo-se os métodos tradicionais de arquivamento em bases de dados (árvores balanceadas, etc.).

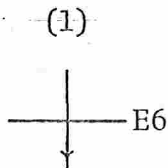
Note-se o aparecimento do operador η introduzido no capítulo 1. Leia-se $\eta\underline{r}y{:}Q(y)$ como "escolha-se algum y do tipo r tal que Q(y) seja verdadeiro".

<u>Prova de E16:</u>

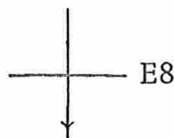
(1) <u>consulta</u> $C(\underline{R}X)\underline{R}$: $\{\underline{r}y : y \in X \wedge P(y)\}$

Mudando-se a nomenclatura e aplicando-se o esquema E6 (capítulo 2) à qualificação:

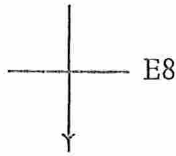
(1)



(2) <u>função</u> $C(\underline{R}X)\underline{R}$: $\eta\underline{R}Y$: $Y = \{\underline{r}y : y \in X \wedge P(y)\}$

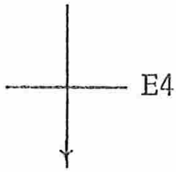


(3) <u>função</u> $C(\underline{R}X)\underline{R}$: $\eta\underline{R}Y$: $(X=\emptyset \wedge X\neq\emptyset) \wedge Y=\{\underline{r}y : y \in X \wedge P(y)\}$



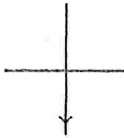
(4) <u>função</u> $C(\underline{R}X)\underline{R}$:

$\eta\underline{R}Y\colon X=\emptyset \wedge Y=\{\underline{r}y : y \in X \wedge P(y)\}$
$\vee X\neq\emptyset \wedge Y=\{\underline{r}y : y \in X \wedge P(y)\}$



(5) <u>função</u> $C(\underline{R}X)\underline{R}$:

<u>se</u> $X=\emptyset$ <u>então</u> $\{\underline{r}y : y \in X \wedge P(y)\}$
<u>senão</u> $\{\underline{r}y : y \in X \wedge P(y)\}$

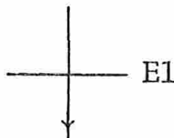


(6) <u>função</u> $C(\underline{R}X)\underline{R}$:

<u>se</u> $X=\emptyset$ <u>então</u> $\emptyset$ <u>senão</u> $\{\underline{r}y : y \in X \wedge P(y)\}$

Neste ponto introduzimos uma novidade nas deduções feitas até aqui. Na cláusula "senão" da seleção acima, temos a garantia de que $\exists \underline{r}y : y \in X$ pois $X\neq\emptyset$. Neste caso, aplicando E1 (capítulo 1), temos:

(6)



(7) <u>função</u> $C(\underline{R}X)\underline{R}$:

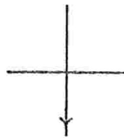
<u>se</u> $X=\emptyset$ <u>então</u> $\emptyset$ <u>senão</u> $\ulcorner \eta\underline{r}y' : y' \in X;$
$\{\underline{r}y : y \in X \wedge P(y)\} \lrcorner$

<u>função</u> C(<u>R</u>X)<u>R</u>:

(8) <u>se</u> X=∅ então ∅ <u>senão</u> ⌈η<u>r</u>y':y'∈X;

{<u>r</u>y:(P(y')∨¬P(y'))∧y∈X∧P(y)}⌋

Aplicando passos exatamente análogos à passagens de (3) a (6) obtemos:



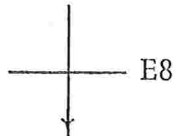
<u>função</u> C(<u>R</u>X)<u>R</u>:

<u>se</u> X=∅ <u>então</u> ∅

(9) <u>senão</u> ⌈η<u>r</u>y':y'∈X;

<u>se</u> P(y') <u>então</u> {<u>r</u>y:y∈X∧P(y)}

<u>senão</u> {<u>r</u>y:y∈X∧P(y)}⌋



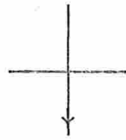
<u>função</u> C(<u>R</u>X)<u>R</u>:

se X=∅ <u>então</u> ∅

(10) <u>senão</u> ⌈η<u>r</u>y':y'∈X;

<u>se</u> P(y') <u>então</u> {<u>r</u>y:(y=y'∨y≠y')∧y∈X∧P(y)}

<u>senão</u> {<u>r</u>y:(y=y'∨y≠y')∧y∈X∧P(y)}⌋



função C(RX)R:

se $X=\emptyset$ então $\emptyset$

(11) senão ⌈ηry' : y'∈X;

se P(y') então {ry:y=y'∧y∈X∧P(y)∨
y≠y'∧y∈X∧P(y)}

senão {ry:y=y'∧y∈X∧P(y)∨
y≠y'∧y∈X∧P(y)}

função C(RX)R:

se $X=\emptyset$ então $\emptyset$

senão ⌈ηry' : y'∈X;

(12) se P(y') então {ry:y=y'∧y∈X∧P(y)}
∪{ry:y≠y'∧y∈X∧P(y)}

senão {ry:y=y'∧y∈X∧P(y)}
∪{ry:y≠y'∧y∈X∧P(y)}⌋

Mas, para P(y') verdadeiro, {ry:y=y'∧y∈X∧P(y)}≡{y'} e {ry:y≠y'∧y∈X∧P(y)} ≡ {ry:y∈(X - {y'})∧P(y)}; por outro lado, para P(y') falso, {ry:y=y'∧y∈X∧P(y)} ≡ ∅ e

{ry:y≠y'∧y∈X∧P(y)} ≡ {ry:y∈(X-{y'})∧P(y)};

portanto

(12)

função $C(\underline{R}X)\underline{R}$:

se $X=\emptyset$ então $\emptyset$

(13) senão $\ulcorner \eta \underline{r}y':y'\in X;$

se $P(y')$ então $\{y'\}\cup\{\underline{r}y:y\in(X-\{y'\})\wedge P(y)\}$

senão $\emptyset\cup\{\underline{r}y:y\in(X-\{y'\})\wedge P(y)\}\lrcorner$

função $C(\underline{R}X)\underline{R}$:

se $X=\emptyset$ então $\emptyset$

(14) senão $\ulcorner \eta \underline{r}y':y'\in X;$

(se $P(y')$ então $\{y'\}$ senão $\emptyset)\cup$

$\{\underline{r}y:y\in(X-\{y'\})\wedge P(y)\}\lrcorner$

Mas note-se que $\{\underline{r}y:y\in(X-\{y'\})\wedge P(y)\}$ é exatamente o resultado do procedimento $C(X-\{y'\})$ conforme a definição de C em (1), portanto, aplicando a contração de E5:

(14)

função $C(\underline{R}X)\underline{R}$:

se $X=\emptyset$ então $\emptyset$

senão $\ulcorner \eta \underline{r}y':y'\in X;$

(se $P(y')$ então $\{y'\}$ senão $\emptyset)\cup C(X-\{y'\})$

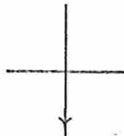
Mudando-se a variável y' para y temos exatamente a forma de E16. Note-se ainda que o algoritmo converge, pois para cada nova chamada de C o argumento diminui até que alcança o conjunto vazio.

#

O resultado é um algoritmo "semi-executável": para que se torne "executável", é necessário substituir o operador de escolha η por uma implementação do mesmo. No entanto, antes de abordarmos esse problema, vamos deduzir um esquema que nos permita eliminar a recursão:

(E17)

(1) <u>consulta</u> C(<u>R</u>X)<u>R</u>: {<u>r</u>y:y∈X∧P(y)}



<u>função</u> C(<u>R</u>X)<u>R</u>:

⌈<u>R</u>X',Y';

(X',Y') ← (X,∅);

<u>enquanto</u> X'≠∅ <u>faça</u> ⟦⌈η<u>r</u>y:y∈X';

(X',Y') ← (X'-{y},Y' ∪ <u>se</u> P(y)

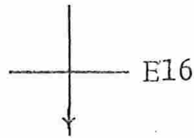
<u>então</u> {y}

<u>senão</u> ∅)⌋⟧;

Y'⌋

<u>Prova</u>:

(1)



(2)

```
função C(RX)R:
        se X=∅ então ∅
               senão ⌈ηry:y∈X;
                      (se P(y) então {y} senão ∅)∪C(X-{y}) ⌋
```

Apliquemos o seguinte esquema introduzido por Bauer et al. [10]:

(E18)

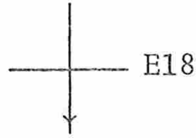
```
função F(tx)t':
        ⌈R;
         se B então ⟦⌈S;
                     F(x')σx''⌋⟧
               senão ⟦⌈T;
                     c⌋⟧
                       ⌋
```

1) não há outras chamadas diretas ou indiretas de $\overline{F}$
2) ∀a,b,c:(aσb)σc = (aσc)σb
3) c constante do tipo t'

```
função F(tx)t': G(x,c)
função G(tx,t'y)t':
        ⌈R;
         se B então ⟦⌈S;
                     G(x',yσx'')⌋⟧
               senão ⟦⌈T;
                     y⌋⟧
                       ⌋
```

Como ∪ é comutativo, e trocando os termos do então... e do senão... de X=∅ (usando-se para isso X≠∅):

(2)



<u>função</u> C(<u>R</u>X)<u>R</u>: C'(X,∅);

<u>função</u> C'(<u>R</u>X,Y)<u>R</u>:

(3)

⌈se X≠∅ <u>então</u> ⟦⌈ η<u>r</u>y:y∈X;

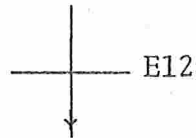
C'(X-{y},Y ∪ <u>se</u> P(y) <u>então</u> {y}

<u>senão</u> ∅⌋⟧

<u>senão</u> Y⌋

Note-se que E7 (capítulo 2) é um caso particular de E18 quando F(x')σx" = F(x'). Agora temos uma forma que permite a eliminação da recursão:

(3)



<u>função</u> C(<u>R</u>X)<u>R</u>: C'(X,∅);

<u>função</u> C'(<u>R</u>X,Y)<u>R</u>:

(4)

⌈<u>R</u>X',Y';

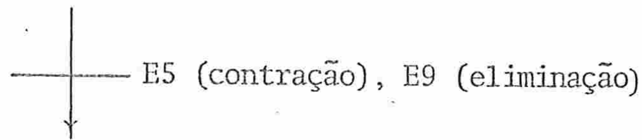
(X',Y') ⟵ (X,Y);

<u>enquanto</u> X'≠∅ <u>faça</u> ⟦⌈ η<u>r</u>y:y∈X';

(X',Y') ⟵ (X'-{y},Y'∪<u>se</u> P(y) <u>então</u> {y}

<u>senão</u> ∅⌋⟧;

Y'⌋



função C(RX)R:

⌈RX',Y';

(X',Y') ← (X,∅);

(5) enquanto X'≠∅ faça ⌈ ηry:y∈X';

(X',Y') ← (X'-{y},Y' use P(y)

então {y}

senão ∅ ⌋;

Y' ⌋

#

Pelo algoritmo acima, ve-se que uma implementação do operador η de escolha é facilmente realizável já que X é um conjunto finito. Por exemplo, se na implementação do tipo R usar-se um armazenamento sequencial para os elementos do conjunto, η pode sempre selecionar o primento elemento da sequência. No capítulo seguinte daremos um exemplo da implementação dos algoritmos de consulta. O que deve ficar claro neste ponto é que não há dificuldades em projetar-se uma implementação. Veremos, também, que antes de se eliminar a recursão pode-se modificar o algoritmo a fim de aumentar a sua eficiência, ou pode-se deixar essa modificação para um estágio sem recursão.

4.3 - CONSULTAS COM UM QUANTIFICADOR

Vejamos, a seguir, como deduzir algoritmos "semi-executáveis" para consultas mais complexas que F1. Nesta, a

qualificação referia-se somente à n-upla sendo selecionada, e como resultado obtinha-se a n-pla na sua íntegra. No caso das consultas C_1 e C_2 do capítulo anterior, procura-se a projeção das n-plas selecionadas em um de seus domínios. A forma da consulta é, nesse caso:

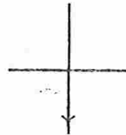
(F2) <u>consulta</u> C(<u>R</u>X)<u>R</u>': {<u>r</u>'y: ∃<u>r</u>x(x∈X∧P(x)∧y= f(x))}

onde <u>R</u> e <u>R</u>' são tipos <u>relação</u>, conjuntos de registros de tipo <u>r</u> e <u>r</u>' respectivamente, e f: <u>r</u> ⟶ <u>r</u>'. f é, em grande parte dos casos, uma função de projeção.

A dedução de algoritmos semi-executáveis para este caso será feita pelos seguintes esquemas de transformação:

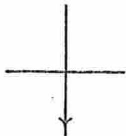
<u>consulta</u> C(<u>R</u>X)<u>R</u>': {<u>r</u>'y: ∃<u>r</u>x(x∈X∧P(x)∧y= f(x))}

(E19)



<u>função</u> C(<u>R</u>X)<u>R</u>':
    <u>se</u> X=∅ <u>então</u> ∅
        <u>senão</u> ⌈ηx:x∈X;
            (<u>se</u> P(x) <u>então</u> {f(x)} <u>senão</u> ∅)∪C(X-{x}) ⌋

(E20)



<u>função</u> C(<u>R</u>X)<u>R</u>':
    ⌈<u>R</u>X',Y';
    (X',Y') ⟵ (X,∅);
    <u>enquanto</u> X'≠∅ <u>faça</u>
        ⟦ η<u>r</u>x:x∈X';
            (X',Y') ⟵ (X'-{x},Y' ∪ <u>se</u> P(x) <u>então</u> {f(x)} senão ∅⟧;
    Y'⌋

Note-se a semelhança com E16 e E17: no lugar de {y} aparece simplesmente {f(x)}. Isso vem de certa maneira mostrar que o quantificador existencial não influencia essencialmente a consulta, servindo apenas para não colocarmos a função f na especificação do elemento do conjunto que estamos procurando, como mostramos na observação (7) do capítulo anterior. De fato, esses esquemas reduzem-se aos anteriores se $f(x) = x$.

Não daremos aqui as provas desses dois esquemas: a de E19 segue passos exatamente análogos à de E16, com as seguintes ressalvas: correspondendo a (5)-(6) de E16 teríamos, $X = \emptyset \Longrightarrow \{\underline{r}'y : \exists \underline{r}x(x \in X \wedge P(x) \wedge y = f(x)\} = \emptyset$; em (11)-(12) é preciso aplicar a regra $\exists a(Q(a) \vee Q'(a)) \equiv \exists a(Q(a)) \vee \exists a(Q'(a))$; Já E20 é idêntica a E17, a menos de {f(x)} que substitui {y}.

Com os esquemas acima podemos eliminar o quantificador existencial que aparece no início (da esquerda para a direita) das qualificações, como por exemplo, não só C_1 e C_2 (do capítulo anterior) que já estão na forma adequada, mas também C_4, etc.

4.4 - CONSULTAS COM DOIS QUANTIFICADORES EXISTENCIAIS

A consulta C_4 leva-nos à próxima forma de consulta: aquela em que alguma <qualif> (veja sintaxe no capítulo anterior) da qualificação não é uma simples sentença do cálculo proposicional, ou contém um quantificador existencial devido a uma projeção, mas sim contém relacionamentos entre n-plas. Vejamos primeiramente o caso de C_4, isto é, esse relacionamento se expressa por meio de quantificador existencial. Teremos a seguinte forma:

(F3) $\underline{\text{consulta}}\ C(\underline{R}X, \underline{R}'Y)\underline{R}''$:

$$\{\underline{r}''z : \exists \underline{r}x(x \in X \wedge z = f(x) \wedge P(x) \wedge \exists \underline{r}'y(y \in Y \wedge P'(x,y)))\}$$

onde $\underline{R}$, $\underline{R}'$ e $\underline{R}''$ são tipos-relação, conjuntos de registros com estrutura de tipos $\underline{r}$, $\underline{r}'$ e $\underline{r}''$ respectivamente. O conectivo entre P(x) e $\exists r'y$ é $\wedge$ pois trata-se de condição adicional envolvendo uma n-pla y a mais.

Note-se que em C_4 temos X = Y = E; F3 é portanto uma generalização, no entanto, ao aplicá-la em C_4 deverão ser usados ambos os parâmetros por uma razão que ficará clara posteriormente.

Para esta consulta, pode-se aplicar os esquemas E19 e E20 colocando-se em lugar de P(x) desses esquemas o termo composto $P(x)\wedge\exists\underline{r}'y(y\in Y\wedge P'(x,y))$ de F3. O passo seguinte seria transformar em um algoritmo "semi-executável" o termo contendo esse último quantificador. Para isso, teríamos as seguintes versões, respectivamente recursiva e iterativa:

(F3)

↓ E19 (tomando-se $\underline{R}$ como tipo composto)

```
     função C(RX,R'X)R":
           se X=∅ então ∅
(1)              senão ⌈ηrx:x∈X;
                        (se P(x)∧∃r'y(y∈Y∧P'(x,y))
                            então {f(x)} senão ∅)∪C(X-{x},Y)⌋
```

↓ E20

```
     função C(RX,R'Y)R":
         ⌈R,X',Z';
          (X',Z') ← (X,∅);
          enquanto X'≠∅ faça
(2)             ⟦ηrx:x∈X';
                 (X',Z') ← (X'-{x}, Z' ∪ se P(x)∧∃r'y(y∈Y∧P'(x,y))
                                              então {f(x)} senão ∅) ⟧;
          Z' ⌋
```

Observando-se os dois algoritmos acima, notamos que, de ambos, podemos extrair a seguinte forma:

```
     função C(RX,R'Y)R":
          ⌈
          :
(3)       ‖⌈ηrx:x∈X';
            ... ∃r'y(y∈Y∧P'(x,y))... ⌋‖
          :
          ⌋
```

em que X' pode ser o próprio X, como é o caso de (1).

Para eliminar o quantificador, vamos criar um novo procedimento, que retorna um valor lógico (booleano):

```
                          (3)
                           |
                       ----+---- E9
                           ↓
     função C(RX,R'Y)R":
          ⌈:
          :
(4)       ⌈função B(rx',R'Y') log:∃r'y'(y'∈Y'∧P'(x',y'));
           ηrx:x∈X';
           ... ∃r'y(y∈Y∧P'(x,y))... ⌋‖
          :
          ⌋
                           |
                       ----+---- E5 (contração); mudança de identificado-
                           ↓                                  res de B
     função C(RX,R'Y)R":
          ⌈:
          :
          ‖⌈função B(rx,R'Y) log: ∃r'y(y∈Y∧P(x,y));
(5)        ηrx:x∈X';
           ...B(x,Y)... ⌋‖
          :
          ⌋
```

Portanto nosso problema consiste agora em achar um algoritmo "semi-executável" para a função B. Temos

(E21) função B(rx,R'Y) log: ∃r'y(y∈Y∧P(x,y))

registro (...)r,(...)r'
relação r'R'

função B(rx,R'Y) log:

se Y=∅ então falso

senão ⌈ηr'y:y∈Y;

se P(x,y) então verdadeiro

senão B(x,Y-{y})⌋

(E22)

função B(rx,R'Y) log:

⌈RY'; log b;

(Y',b) ← (Y, falso);

enquanto Y'≠∅∧ ¬b faça ‖⌈ηr'y:y∈Y';

(Y',b) ← (Y'-{y},P(x,y)) ⌋‖;

b⌋

PROVA DE E21

Como as técnicas de transformações foram bem exemplificadas nos capítulos 1 e 2, e alguns passos são semelhantes à prova de E16, não exporemos todos os passos intermediários. Aliás, é interessante acompanhar-se a prova de E16.

(1) função $B(\underline{r}x,\underline{R}'Y)$ log: $\exists\underline{r}'y(y\in Y\wedge P(x,y))$

(2) função $B(\underline{r}x,\underline{R}'Y)$ log: $\exists\underline{r}'y((Y=\emptyset\vee Y\neq\emptyset)\wedge y\in Y\wedge P(x,y))$

(3) função $B(\underline{r}x,\underline{R}'Y)$ log:

se $Y=\emptyset$ então $\exists\underline{r}'y(y\in Y\wedge P(x,y))$

senão $\exists\underline{r}'y(y\in Y\wedge P(x,y))$

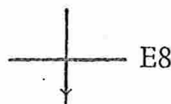
(4) função $B(\underline{r}x,\underline{R}'Y)$ log:

se $Y=\emptyset$ então falso

senão $\ulcorner\eta\underline{r}'z{:}\,z\in Y;$

$(P(x,z)\vee\neg P(x,z))\wedge\exists\underline{r}'y((z=y\vee z\neq y)\wedge y\in Y\wedge$

$P(x,y))\lrcorner$



(5) função $B(\underline{r}x,\underline{R}'Y)$ log:

se $Y=\emptyset$ então falso

senão $\ulcorner\eta\underline{r}'z{:}\,z\in Y;$

$P(x,z)\wedge\exists\underline{r}'y(z=y\wedge y\in Y\wedge P(x,y))\vee$

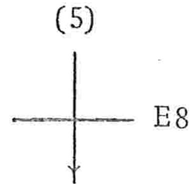
$P(x,z)\wedge\exists\underline{r}'y(z\neq y\wedge y\in Y\wedge P(x,y))\vee$

$\neg P(x,z)\wedge\exists\underline{r}'y(z=y\wedge y\in Y\wedge P(x,y))\vee$

$\neg P(x,z)\wedge\exists\underline{r}'y(z\neq y\wedge y\in Y\wedge P(x,y))\lrcorner$

Mas em nosso caso, $\exists \underline{r}'y(z=y \wedge y \in Y \wedge P(x,y)) \equiv P(x,z)$ e, por outro lado, $\exists \underline{r}'y(z \neq y \wedge y \in Y \wedge P(x,y)) \equiv \exists \underline{r}'y(y \in Y-\{z\} \wedge P(x,y))$. Temos, então:

(5)



<u>função</u> B(...

⋮

<u>senão</u> $\ulcorner \eta \underline{r}' z: z \in Y;$

(6) $P(x,z) \wedge P(x,z) \vee$

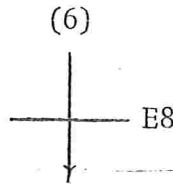
$P(x,z) \wedge \exists \underline{r}'y(y \in (Y-\{z\}) \wedge P(x,y)) \vee$

$\neg P(x,z) \wedge P(x,z) \vee$

$\neg P(x,z) \wedge \exists \underline{r}'y(y \in (Y-\{z\}) \wedge P(x,y)) \lrcorner$

como $A \vee A \wedge B \equiv A$, temos:

(6)



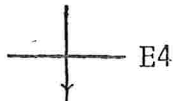
<u>função</u> B(...

⋮

(7) <u>senão</u> $\ulcorner \eta \underline{r}' z: z \in Y;$

$P(x,z) \vee$

$\neg P(x,z) \wedge \exists \underline{r}'y(y \in (Y-\{z\}) \wedge P(x,y)) \lrcorner$



<u>função</u> B(...

⋮

(8) <u>senão</u> $\ulcorner \eta \underline{r}' z: z \in Y;$

se $P(x,z)$ <u>então</u> <u>verdadeiro</u>

<u>senão</u> $\exists \underline{r}'y(y \in (Y-\{z\}) \wedge P(x,y)) \lrcorner$

```
     função B(rx,R'Y) log:
            se Y=∅ então ∅
(9)                senão ⌈ηr'z:z∈Y;
                          se P(x,z) então verdadeiro
                                    senão B(x,Y-{z}) ⌋
```

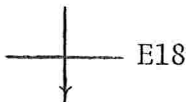
Trocando-se o identificador z por y obtemos a forma de E21. A "parada" da recursão é garantida, pois para cada nova chamada recursiva de B a cardinalidade do 2º parâmetro Y diminui, e para Y = ∅ não há mais chamada recursiva.

#

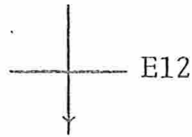
Prova de E22

```
     função B(rx,R'Y) log:
            se Y=∅ então falso
(1)                senão ⌈ηr'y:y∈Y;
                          se P(x,y) então verdadeiro
                                    senão B(x,Y-{y} ⌋
```

```
     função B(rx,R'Y) log:
            se Y≠∅ então ⌈ηr'y:y∈Y;
(2)                          P(x,y)vB(x,Y-{y}) ⌋
                   senão falso
```




```
     função B(rx,R'Y) log: B'(x,Y,falso);
     função B'(rx,R'Y,log b):
(3)         ⌈se Y≠∅ então ‖⌈ηr'y:y∈Y;
                             B'(x,Y-{y},bvP(x,y))⌋‖
                   senão b ⌋
```



<u>função</u> B(<u>r</u>x,<u>R</u>'Y) <u>log</u>: B'(x,Y, <u>falso</u>);
<u>função</u> B'(<u>r</u>x',<u>R</u>'<u>Y</u>',<u>log</u> b'):
⌈<u>r</u>x;<u>R</u>Y;<u>log</u> b;
(4) (x,Y,b) ← (x',Y',b');
<u>enquanto</u> Y≠∅ <u>faça</u> ‖⌈η<u>r</u>'y:y∈Y;
(x,Y,b) ← (x,Y-{y},b∨P(x,y)) ‖;
b⌋

Como, pela definição de comando de atribuição (veja por exemplo [19])

(E23) $(a,b) \leftarrow (e_1,e_2)$

a não ocorre em e_2, e_1 sem efeitos colaterais nas variáveis de e_2

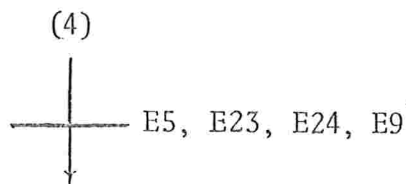
$a \leftarrow e_1;$
$b \leftarrow e_2;$

e

ε

(E24)

$a \leftarrow a$

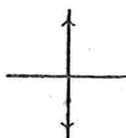



```
     função B(rx,R'Y) log:
        ⌈RY'; log b;
         (Y',b) ← (Y, falso);
(5)      enquanto Y'≠∅ faça ⌈⌈ηr'y:y∈Y';
                              Y' ← Y'-{y};
                              b ← b∨P(x,y) ⌋⌋;
        b⌋
```

Para aumentar a eficiência, podemos interromper a iteração assim que b se torna verdadeiro, pois daí em diante ele não muda de valor, independentemente do valor de Y e de P. Para obter a nova versão, aplicamos o esquema:

```
(E25)        ⌈log b;
              S1;
              b ← falso;
              enquanto B faça ⌈⌈S2; b ← b∨e ⌋⌋;
              b⌋
```

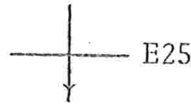
valor de b não alterado por S_2; S_2 e B sem efeitos colaterais fora de ⌈ ... ⌋



```
              ⌈log b;
               S1;
               b ← falso;
               enquanto B∧¬b faça ⌈⌈S2; b ← e⌋⌋;
               b⌋
```

Temos, então,

(5)




```
      função B(rx,R'Y) log:
              ⌈R'Y'; log b;
               (Y',b) ← (Y, falso);
(6)            enquanto Y'≠∅∧¬b faça ⌈⌈ηr'y:y∈Y';
                                      Y ← Y-{y};
                                      b ← P(x,y) ⌋⌋;
               b⌋
```

Aplicando E28 a (6), obtemos a forma procurada de E22.

#

Agora poderemos dar os algoritmos "semi-executáveis" finais para F3; compondo (5) de F3 com E21 temos:

(E26) consulta C(RX,R'Y)R'':

$\{r''z:\ \exists rx(x\in X\wedge z=f(x)\wedge P(x)\wedge\exists r'y(y\in Y\wedge P'(x,y)))\}$

```
     função C(RX,R'Y)R'':
         se X=∅
         então ∅
         senão ⌈função B(rx,R'Y) log:
                    se Y=∅
                      então falso
(1)                   senão ⌈⌈η r'y:y∈Y;
                               se P'(x,y) então verdadeiro
                                          senão B(x,Y-{y})⌋⌋;
               ηrx:x∈X;
               (se P(x)∧B(x,Y)
                  então {f(x)}
                  senão ∅)∪C(X-{x},Y)⌋
```

Na versão não-recursiva, podemos aplicar E22, depois substituir a chamada de B pela sua expansão segundo E5, eliminar posteriormente a função B aplicando E9 e, aplicando finalmente E23 para maior clareza de notação:

(E27) (1)

```
função C(RX,R'Y)R'':
  ⌈RX';R''Z;
   (X',Z) ← (X,∅);
   enquanto X'≠∅ faça
     ‖⌈ηrx:x∈X';
       X' ← X'-{x};
       Z ← Z ∪ se P(x)∧⌈R'y'; log b;
                         (Y',b) ← (Y, falso);
                         enquanto Y'≠∅∧ ¬b faça
                           {⌈ηr'y:y∈Y';
                             (Y',b) ← (Y'-{y},P'(x,y)) ⌋};
                         b⌋ então {f(x)} senão ∅ ‖;
  Z⌋
```

Nesta altura podemos justificar uma afirmação que fizemos quando da exposição de F3 (pág. 70), isto é, que X e Y de F3 (e portanto de E26 e E27) reduzem-se a uma só relação, E, no caso de consultas como C_4 do capítulo anterior. No entanto, como afirmamos, devem ser usados ambos os parâmetros de F3, E26 e E27. A razão disso é a seguinte: se em E26, no caso de se ter X = Y, reduzíssemos a um os dois parâmetros formais de C, cada vez que B fosse chamado dentro de uma chamada recursiva de C, a relação Y (que agora seria a própria X) não seria a mesma relação original quando da pri

meira chamada (não-recursiva) de C, isto é, não estaríamos verificando a existência de algum $y \in X$ tal que $P'(x,y)$ e sim algum $y \in X'$, onde X' é obtido de X excluindo-se as n-plas x já selecionadas em $\eta \underline{r} x{:}x \in X$. Em E27 essa razão torna-se ainda mais óbvia, se em lugar de Y em $(Y',b) \leftarrow (Y, \underline{falso})$ se tomasse X' que já não é a mesma relação X original. O processamento seria correto, no entanto, tomando-se nessa atribuição X em lugar de Y, pois X não é alterada durante todo o processamento. No próximo capítulo, ao desenvolver um programa para a consulta C_4, exemplificaremos a correta utilização de E26 com dois parâmetros, no caso de se ter uma só relação.

Finalmente, observe-se que em F3 o conectivo $\wedge$ da expressão $P(x) \wedge \exists \underline{r}'y(\ldots)$ permanece inalterado em E26 e E27, onde o predicado $\exists \wedge y(\ldots)$ é substituido por $B(x,Y)$ e pelo bloco $\ulcorner \ldots \lrcorner$, respectivamente. Além disso, esse conectivo não influiu na dedução. Assim, sendo, pode ser substituído por qualquer conectivo lógico diádico: $\vee$, $\Rightarrow$, $\equiv$, $\neg\equiv$, etc.

4.5 - CONSULTAS COM QUANTIFICADORES EXISTENCIAIS

Continuando o desenvolvimento de esquemas de transformações de consultas formuladas em cálculo de predicados para algoritmos semi-executáveis, temos consultas com um número qualquer de quantificadores existenciais, que assumem a seguinte forma geral:

$$(F4)\ \underline{\text{consulta}}\ C(\underline{R}X,\underline{R}'X',\ldots,\underline{R}^{(n)}X^{(n)})\underline{R}^{(n+1)}$$

$$\{\underline{r}^{(n+1)}z{:}\exists \underline{r}x(z=f(x)\wedge x \in X \wedge P(x)\alpha' \exists \underline{r}'x'(x' \in X' \wedge P'(x,x')\alpha''$$

$$\exists r''x''(x'' \in X'' \wedge P''(x,x',x'')\alpha''' \exists r'''x'''(\ldots\alpha^{(n)}$$

$$\exists \underline{r}^{(n)}x^{(n)}(x^{(n)} \in X^{(n)} \wedge P^{(n)}(x,x',x'',\ldots,x^{(n)})\ldots))))\}$$

onde $R^{(i)}$ é uma relação de n-plas de tipo $r^{(i)}$ e $P^{(i)}$ sentenças do cálculo proposicional (isto é, sem quantificadores), onde $\alpha^{(i)}$ é um conectivo lógico diádico qualquer ($\vee$, $\wedge$, $\Longrightarrow$, $\Longleftrightarrow$, $\neg\equiv$, etc.), $i=1,\ldots,n$. Uma consulta dessa forma pode ser encarada como uma extensão da forma F3 e os esquemas de transformação podem ser aplicados quase que diretamente. De fato, observa-se que o primeiro quantificador existencial (mais externo) encaixa-se exatamente na forma F2, em que $\alpha' = \wedge$, estendendo-se P(x) de F2 para conter ainda a "conexão" α' com todo o termo que constitui o escopo de $\exists r'x'$. Observando-se E26 e E27, e comparando-se-os com E19 e E20, vemos que também no caso de F4 teremos uma parte "externa" semelhante a todos esses esquemas:

(E28) (F4)

<u>função</u> C(<u>R</u>X,...,<u>R</u>$^{(n)}$X$^{(n)}$)<u>R</u>$^{(n+1)}$:
  <u>se</u> X=∅ <u>então</u> ∅
(1)   <u>senão</u> ⌈S;
      η<u>r</u>x:x∈X;
      (<u>se</u> P(x)α'B'(x,X')
        <u>então</u>{f(x)}senão ∅)∪C(X-{x},X',...,X$^{(n)}$)⌋

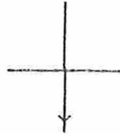
onde S compõem-se da declaração de n funções B',B",...,B$^{(n)}$; cada B(i) tem a forma:

<u>função</u> B$^{(i)}$(<u>r</u>x,<u>r</u>'x',...,<u>r</u>$^{(i-1)}$,x$^{(i-1)}$,<u>R</u>$^{(i)}$X$^{(i)}$)<u>log</u>:
  <u>se</u> X$^{(i)}$=∅
    <u>então</u> <u>falso</u>
    <u>senão</u> ⌈η<u>r</u>$^{(i)}$x$^{(i)}$:x$^{(i)}$∈X$^{(i)}$;
      <u>se</u> P$^{(i)}$(x,x',...,x$^{(i)}$)α$^{(i+1)}$B$^{(i+1)}$(x,x',...,x$^{(i)}$,X$^{(i+1)}$)
        <u>então</u> <u>verdadeiro</u>
        <u>senão</u> B$^{(i)}$(x,x',...,x$^{(i-1)}$,X$^{(i)}$-{x$^{(i)}$})⌋

sendo que para i = n, a expressão condicional entre o segundo se...então... reduz-se a $P^{(n)}(x,x',\ldots,x^{(n)})$, ou, mais formalmente, $\alpha^{(n+1)} = \wedge$, $B^{(n+1)}$ = verdadeiro.

A versão não-recursiva, partindo de (1) de E28 é

(E29) (1)



```
      função C(RX,R'X',...,R(n)X(n))R(n+1):
         ⌈RY,Z;
          (Y,Z) ←— (X,∅);
          enquanto Y≠∅ faça
(2)          ⌈⌈ηrx:x∈Y;
              Y ←— Y-{x};
              Z ←— Z ∪ se P(x)α'B' então {f(x)} senão ∅ ⌋⌋;
         Z ⌋
```

onde B' é um bloco que retorna um valor lógico, da seguinte forma:

```
B' ≡ ⌈R'Y'; log b';
     (Y',b') ←— (X', falso);
     enquanto Y'≠∅ ∧ ¬b' faça
       ⌈⌈ηr'y':y'∈Y';
         (Y',b') ←— (Y'-{y'},P'(x,y')α''B'') ⌋⌋;
     b' ⌋
```

e, indutivamente, $B^{(i)}$ é semelhante a B' substituindo-se em todas as variáveis o índice ' por $^{(i)}$, sendo que em lugar de P', B'' e α'' tem-se $P^{(i)}(x,y',y'',\ldots,y^{(i)})$, $B^{(i+1)}$ e $\alpha^{(i+1)}$ respectivamente.

Se i = n, então não ocorre o termo B na "conexão" α com $P^{(n)}(x,y',\ldots,y^{(n)})$. Evidentemente, $B^{(i)}$ podem ser decla

dos como funções que retornam um valor lógico, para maior clareza.

Tanto E28 como E29 podem ser demonstradas considerando-se cada predicado com um quantificador existencial como uma função, aplicando-se o método usado para demonstração de E26 e E27; notando-se que a expressão $P(x)\wedge\exists \underline{r}'y(\ldots)$ permanece inalterada nos esquemas, havendo apenas substituição de $\exists \underline{r}'y(\ldots)$ pela chamada da função $B(x,Y)$, pode-se substituir $\wedge$ por α.

Antes de abandonarmos esta seção dedicada a quantificadores existenciais notemos que, do cálculo de predicados, esses quantificadores são comutativos e associativos entre si:

$$\exists x\exists yP(x,y) \equiv \exists y\exists xP(x,y) \equiv \exists x(\exists yP(x,y))$$

Assim, consultas formuladas das duas primeiras maneiras podem sempre ser reformuladas, obtendo-se a forma de F4. Lembrando que $\exists y(P(x)\alpha P(x,y))\equiv P(x)\alpha\exists yP(x,y)$ onde y não ocorre em $P(x)$, podemos reescrever sentenças do tipo

$$\exists x\exists y(P(x)\alpha P(x,y)) \text{ como } \exists x(P(x)\alpha\exists yP(x,y))$$

e assim por diante. O teorema das "formas prenex" (vide por exemplo [21]) nos garante a possibilidade de agruparmos convenientemente os quantificadores e os predicados para obter F4.

4.6 - CONSULTAS COM QUANTIFICADORES UNIVERSAIS

Trataremos, a seguir, de consultas com quantificadores universais. Notemos, primeiramente, que em nossa formulação de consultas, o primeiro quantificador que aparece mais à esquerda é um quantificador existencial; é através dele que é selecionada uma n-pla cuja projeção constituirá u-

ma n-pla da relação-resultado. Quantificadores adicionais, sejam eles existenciais ou universais, só aparecem nos predicados P, P',etc. (ver F1, F2, F3 e F4 deste capítulo,e as consultas C_i do capítulo anterior). Uma consulta análoga a F3 seria:

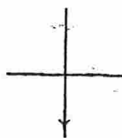
(F5) consulta C(RX,R'Y)R":

$\{\underline{r}''z:\exists\underline{r}x(x\in X\wedge z=f(x)\wedge P(x)\alpha\forall\underline{r}'y(y\in Y\Longrightarrow P'(x,y))\}$

Note-se o aparecimento da implicação lógica no lugar da conjunção de F3. Isso se deve ao fato de que, evidentemente, existem n-plas y do tipo r' que não pertencem a Y. Analogamente a F3, vamos definir uma função cujo valor é o do predicado com o quantificador; obtemos os seguintes esquemas (a notação é idêntica à de E21)

função B(rx,R'Y) log:$\forall\underline{r}'y(y\in Y\Longrightarrow P(x,y))$

(E30)



função B(rx,R'Y) log:
  se Y=∅ então verdadeiro
    senão ⌈ηr'y:y∈Y;
      se P(x,y) então B(x,Y-{x})
        senão falso⌋

(E31)

função B(rx,R'Y) log:
  ⌈RY'; log b';
  (Y',b) ⟵ (Y, verdadeiro);
  enquanto Y'≠∅∧b faça ⌈ηr'y:y∈Y';
      (Y',b) ⟵ (Y'-{y},P(x,y))⌋;
  b⌋

A prova de E30 segue os mesmos passos da prova de E21. Notemos apenas as seguintes diferenças, segundo os passos correspondentes de E21:

- em (3), se $Y=\emptyset$ temos $\forall \underline{r}'y(y\in Y \Rightarrow P'(x,y)) \equiv$ verdadeiro, pois, o antecedente da implicação é sempre falso.

- em (4) teríamos $(P(x,z) \vee \neg P(x,z)) \wedge \forall r'y((z=y \vee z\neq y) \wedge y\in Y \Rightarrow P(x,y))$. Para desenvolver essa sentença, podemos empregar as seguintes propriedades: $(a \vee b) \wedge c \Rightarrow d \equiv (a\wedge c \vee b \wedge c) \Rightarrow d \equiv (a\wedge c \Rightarrow d) \wedge (b \wedge c \Rightarrow d)$, $\forall x(P(x) \wedge Q(x)) \equiv \forall x(P(x)) \wedge \forall x(Q(x))$ e $a \wedge b \wedge c \equiv a \wedge b \wedge a \wedge c$, obtendo-se

- em (5), $\ulcorner \eta \underline{r}' z : z\in Y;$

$$P(x,z) \wedge \forall \underline{r}'y(z=y \wedge y\in Y \Rightarrow P(x,y)) \wedge$$
$$P(x,z) \wedge \forall \underline{r}'y(z\neq y \wedge y\in Y \Rightarrow P(x,y)) \vee$$
$$\neg P(x,z) \wedge \forall \underline{r}'y(z=y \wedge y\in Y \Rightarrow P(x,y)) \wedge$$
$$\neg P(x,z) \wedge \forall \underline{r}'y(z\neq y \wedge y\in Y \Rightarrow P(x,y)) \lrcorner$$

Mas $\forall \underline{r}'y(z=y \wedge y\in Y \Rightarrow P(x,y)) \equiv P(x,z)$ pois o antecedente da implicação somente é verdadeiro para $y = z$;

$z\neq y \wedge y\in Y \equiv y\in Y-\{z\}$; $\neg P(x,y) \wedge P(x,y) \equiv$ falso, obtendo-se:

- (7) $\ulcorner \eta \underline{r}' z : z\in Y;$

$$P(x,z) \wedge \forall \underline{r}'y(y\in Y-\{z\} \Rightarrow P(x,y)) \vee \neg P(x,y) \wedge \text{falso} \lrcorner$$

Seguem-se os correspondentes a (8) e, finalmente a (9) que é a forma procurada.

A demonstração de E31 é exatamente análoga à de E22.

Note-se que a simetria de E30 e E31 com E21 e E22 é quase perfeita. Ela não é totalmente perfeita pois a proposição $y\in Y \Rightarrow P(x,y)$ é diferente de $y\in Y \wedge P(x,y)$. Se essas proposições fossem equivalentes, esperar-se-ía uma perfeita simetria devido às propriedades $\forall x(P(x)) \equiv \neg \exists x(\neg P(x))$ e $\exists x(P(x)) \equiv \neg \forall x(\neg P(x))$.

A generalização para uma consulta com uma sequên-

cia de quantificadores universais, análoga a F4 seria

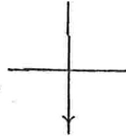
(F6) $\underline{\text{consulta}}\ C(\underline{R}X,\underline{R}'X',\ldots,\underline{R}^{(n)}X^{(n)})\underline{R}^{(n+1)}$

$\{\underline{r}^{(n+1)}z:\ \exists\underline{r}x(z=f(x)\wedge x\in X\wedge P(x)\alpha'\forall\underline{r}'x'(x'\in X'\Rightarrow P'(x,x')\alpha''$

$\forall\underline{r}''x''(x''\in X''\Rightarrow P''(x,x',x'')\alpha'''\forall\underline{r}'''x'''(\ldots\alpha^{(n)}\forall\underline{r}^{(n)}x^{(n)}$

$(x^{(n)}\in X^{(n)}\Rightarrow P^{(n)}(x,x',x'',\ldots,x^{(n)})\ldots)))))\}$

(E32)



$\underline{\text{função}}\ C(\underline{R}X,\ldots,\underline{R}^{(n)}X^{(n)})\underline{R}^{(n+1)}$

<<idêntica a (1) de E28>>

Neste caso, $B^{(i)}$ de S de (1) de E28 assumem a forma:

$\underline{\text{função}}\ B^{(i)}(\underline{r}x,\underline{r}'x',\ldots,\underline{r}^{(i-1)}x^{(i-1)},\underline{R}^{(i)}X^{(i)})\ \underline{\text{log}}$:

$\underline{\text{se}}\ X^{(i)}=\emptyset$

$\underline{\text{então}}\ \underline{\text{verdadeiro}}$

$\underline{\text{senão}}\ \ulcorner\eta\underline{r}^{(i)}x^{(i)}:x^{(i)}\in X^{(i)};$

$\underline{\text{se}}\ P^{(i)}(x,x',\ldots,x^{(i)})\ B^{(i+1)}(x,x',\ldots,x^{(i)}X^{(i+1)})$

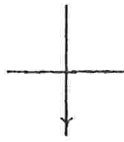
$\underline{\text{então}}\ B^{(i)}(x,x',\ldots,x^{(i-1)},X^{(i)}-\{x^{(i)}\})$

$\underline{\text{senão}}\ \underline{\text{falso}}\ \lrcorner$

sendo que. para $i=n$, a expressão condicional entre o segundo <u>se</u>... <u>então</u> reduz-se a $P^{(n)}(x,x',\ldots,x^{(n)})$.

A forma não-recursiva é neste caso:

(E33)



<u>função</u> $C(\underline{R}X,\underline{R}'X',\ldots,\underline{R}^{(n)}X^{(n)})\underline{R}^{(n+1)}$:

<<idêntica a E29>>

onde B' é, agora,

B' ≡ ⌈R'Y'; <u>log</u> b';

(Y',b') ← (X', <u>verdadeiro</u>);

<u>enquanto</u> $Y'\neq\emptyset\wedge b'$ <u>faça</u>

‖⌈$\eta\underline{r}'y':y'\in Y'$;

$(Y',b') \leftarrow (Y'-\{y'\},P'(x,y')\wedge B'')$ ⌋‖;

b'⌋

e, indutivamente, $B^{(i)}$ é obtida como em E29.

4.7 - <u>CONSULTAS COM AMBOS QUANTIFICADORES</u>

Finalmente, devemos examinar o caso em que em uma consulta aparecem ambos os quantificadores; na forma abaixo, $\theta^{(i)}\in\{\forall,\exists\}$ e $\rho^{(i)}\in\{\Rightarrow,\wedge\}$; se $\theta^{(i)}=\forall$, $\rho^{(i)}=\Rightarrow$; se $\theta^{(i)}=\exists$, $\rho^{(i)}=$ $=\wedge$; $\alpha^{(i)}$, representa, como anteriormente, um conectivo lógico diádico qualquer.

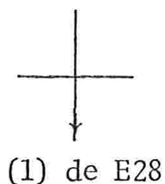
(F7) <u>consulta</u> $C(\underline{R}X,\underline{R}'X',\ldots,\underline{R}^{(n)}X^{(n)})\underline{R}^{n+1}$:

$\{\underline{r}^{(n+1)}z:\exists\underline{r}x(z=f(x)\wedge x\in X\wedge P(x)\alpha'\theta'\underline{r}'x'(x'\in X'\rho'P'(x,x')\alpha''$

$\theta''\underline{r}''x''(x''\in X''\rho''P''(x,x',x'')\alpha'''\theta'''\underline{r}'''x'''(\ldots\alpha^{(n)}$

$\theta^{(n)}\underline{r}^{(n)}x^{(n)}(x^{(n)}\in X^{(n)}\rho^{(n)}P^{(n)}(x,x',\ldots,x^{(n)}))\ldots)))))\}$

(E34)



(1) de E28

onde $B^{(i)}$ é declarado como

<u>função</u> $B^{(i)}(\underline{r}x,\underline{r}'x',\ldots,\underline{r}^{(i-1)}x^{(i-1)},\underline{R}^{(i)}X^{(i)})$ <u>log</u>:
<u>se</u> $X^{(i)}=\emptyset$
<u>então</u> $T^{(i)}$
<u>senão</u> $\ulcorner \eta \underline{r}^{(i)}x^{(i)}:x^{(i)}\in X^{(i)}$;
<u>se</u> $P^{(i)}(x,x',\ldots,x^{(i)})\alpha^{(i+1)}B^{(i+1)}(x,x',\ldots,x^{(i)},X^{(i+1)})$
<u>então</u> $V^{(i)}$
<u>senão</u> $W^{(i)} \lrcorner$

onde

- se $\theta^{(i)} = \exists$ então $T^{(i)} =$ <u>falso</u>, $V^{(i)} =$ <u>verdadeiro</u> e
$W^{(i)} = B^{(i)}(x,x',\ldots,x^{(i-1)},X^{(i)}-\{x^{(i)}\})$.
- se $\theta^{(i)} = \forall$ então $T^{(i)} =$ <u>verdadeiro</u>,
$V^{(i)} = B^{(i)}(x,x',\ldots,x^{(i-1)},X^{(i)}-\{x^{(i)}\})$ e
$W^{(i)} =$ <u>falso</u>

e para $i = n$ desaparece o conectivo e o termo $B^{(i+1)}(x,x',\ldots)$

A forma não-recursiva seria, em relação a (E34):

(1)

(E35) <u>função</u> $C(\underline{R}X,\underline{R}'X',\ldots,\underline{R}^{(n)}X^{(n)})\underline{R}^{n+1}$:
<<idêntica a (2) de E29>>

onde

$B' \equiv \ulcorner \underline{R}'Y'$; log b';

$(Y', b') \leftarrow (X', T')$;

enquanto $Y' \neq \emptyset \wedge \beta b'$ faça

$\| \ulcorner \eta \underline{r}'y' : y' \in Y'$;

$(Y', b') \leftarrow (Y' - \{y'\}, P'(x, y') \wedge B'') \rfloor\!\|$;

$b' \lrcorner$

e $B^{(i)}$ como em E29, onde $T^{(i)}$ é o mesmo de (1) de E34, e se $T^{(i)}$ = falso então $\beta = \neg$; se $T^{(i)}$ = verdadeiro então $\beta = \neg\neg$ (ou vazio).

4.8 - CONSULTAS COM NEGAÇÃO DE QUANTIFICADORES

Até este ponto apresentamos formas de consultas sem que nelas ocorresse uma negação de quantificador, como $\neg\exists x(P(x))$ ou $\neg\forall x(P(x))$. Uma solução para esse caso é o de usar as identidades $\neg\exists x(Q(x)) \equiv \forall x(\neg Q(x))$ e $\neg\forall x(Q(x)) \equiv \exists x(\neg Q(x))$. Vejamos como a negação da qualificação da variável ligada x afeta as formas de consultas já vistas.

Em termos do quantificador existencial as qualificações são sempre da forma

$$F \equiv \neg\exists x(x \in X \wedge P(x)\alpha\theta x'(\ldots))$$

Como $\neg(a \wedge b) \equiv \neg a \vee \neg b$,

$$F \equiv \forall x(\neg x \in X \vee \neg(P(x)\alpha\theta x'(\ldots)))$$

Mas $\neg a \vee b \equiv a \Rightarrow b$,

$$F \equiv \forall x(x \in X \Rightarrow \neg(P(x)\alpha\theta x'(\ldots)))$$

Por meio de operações do cálculo proposicional podemos introduzir a negação dentro da expressão $P(x)\alpha\theta x'(\ldots))$, obtendo-se a forma $\bar{P}(x)\bar{\alpha}\beta\theta x'(\ldots)$ onde $\beta \in \{\neg, \neg\neg\}$. Recaímos

assim, no caso anterior em relação a θ se $\theta = \exists$ ou no seguinte, se $\theta = \forall$.

No caso de $\forall$, temos:

$$F \equiv \neg\forall x(x \in X \Rightarrow P(x)\alpha\theta x'(\ldots))$$

Como $\neg(a \Rightarrow b) \equiv a \wedge \neg b$, temos

$$F \equiv \exists x(x \in X \wedge \neg(P(x)\alpha\theta x'(\ldots))$$

Com essa solução para o problema de negação de quantificadores, podemos resolve-lo aplicando transformações lógicas diretamente nas consultas. Uma outra solução é a de se trabalhar ao nível dos algoritmos deduzidos.

Examinando-se F7, E34 e E35 (e portanto E28 e E29), notamos que os predicados com quantificadores são transformados em funções e blocos. A negação de um desses predicados é portanto trivialmente transformada na negação da chamada da função correspondente ou do bloco. Com isso, podemos finalmente formular a forma seguinte, que é a mais geral possível, dentro do tipo de consultas abordado neste trabalho, em que $\beta^{(i)}$ é ou $\neg$ ou vazio:

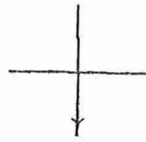
(F8) <u>consulta</u> $C(\underline{R}X,\underline{R}'X',\ldots,\underline{R}^{(n)}X^{(n)})\underline{R}^{n+1}$:

$$\{\underline{r}^{(n+1)}z:\exists\underline{r}x(z=f(x)\wedge x\in X\wedge P(x)\alpha'\beta'\theta'\underline{r}'x'(x'\in X'\rho'P'(x,x')\alpha''$$
$$\beta''\theta''\underline{r}''x''(x''\in X''\rho''\ P''(x,x',x'')\alpha'''\beta'''\theta'''\underline{r}'''x(\ldots\alpha^{(n)}$$
$$\beta^{(n)}\theta^{(n)}\underline{r}^{(n)}x^{(n)}(x^{(n)}\in X^{(n)}\rho^{(n)}P^{(n)}(x,x',\ldots,x^{(n)}))\ldots)))))\}$$

(E36)

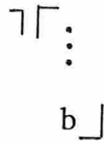
(idêntico a E34, fazendo-se a chamada de $B^{(i)}$ ser precedida de $\beta^{(i)}$)

(E37)

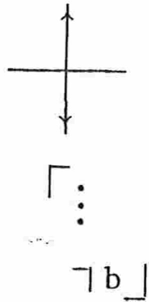


(idêntico a E35, colocando-se $\beta^{(i)}b^{(i)}$ no lugar de $b^{(i)}$ no fim do bloco $B^{(i)}$)

pois, evidentemente,



(E38)



- ¤ -

CAPÍTULO 5

APLICAÇÕES DOS ESQUEMAS DE TRANSFORMAÇÕES DE CONSULTAS A BASES DE DADOS

5.1 - INTRODUÇÃO

Neste capítulo desenvolveremos um exemplo de aplicação dos esquemas de transformações do capítulo anterior, deduzindo um algoritmo "semi-executável" que realiza uma das consultas exemplificadas no capítulo 3. Mostraremos também, como implementar algoritmos para realizarem a escolha determinada pelo operador η, derivando programas "executáveis". Veremos, além disso, como se pode otimizar o programa de consulta obtido. Não apresentaremos a versão recursiva já que, sendo os parâmetros das consultas conjuntos, o uso de procedimentos recursivos é proibitivo, pois para cada chamada seria necessário guardar-se cada conjunto nos registros de ativação da pilha de tempo — objeto.

5.2 - EXEMPLO DE CONSULTA

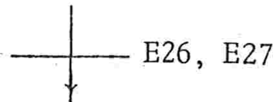
Tomemos como exemplo a consulta C_4 do capítulo 3. Para maior clareza, vamos formulá-la novamente:

```
     consulta C4 (EMPRESA E) relação (cadeia N, int SAL):
        {(cadeia N, int SAL): ∃FUNCIONÁRIO G(G∈E∧G•nome-do-funcioná-
(1)               rio=G•salário∧∃FUNCIONÁRIO∧F(F∈E F•nome-do-gerente=
                  G•nome-do-funcionário))}
```

Já que essa consulta contém dois quantificadores existenciais, empregaremos E26 e E27. Como eles exigem dois parâmetros, façamos inicialmente uma transformação que amplia o número de parâmetros análogo à passagem de (5) para (6) do capítulo 2:

(1)

```
     consulta C4 (EMPRESA E) relação (cadeia N, int SAL):C'4 (E,E);
     consulta C'4 (EMPRESA X,Y) relação (cadeia N, int SAL):
(2)     {(cadeia N, int SAL):∃FUNCIONÁRIO G(G∈X∧N=G•nome-do-funcioná-
                  rio SAL=G•salário∧∃FUNCIONÁRIO F(F∈Y∧F•nome-do geren
                  te=G•nome-do-funcionário))}
```




```
     função C4 (EMPRESA E) relação (cadeia N, int SAL):C'4 (E,E);
     função C'4 (EMPRESA X,Y) relação (cadeia N, int SAL):
               ⌈EMPRESA E'; relação (cadeia N, int SAL) Z;
                (E',Z) ← (X,∅);
                enquanto E'=∅ então
                   ⌈⌈η FUNCIONÁRIO G:G∈E';
                     E' ← E'-{G};
(3)                  Z ← Z use ⌈EMPRESA E"; log b;
                               (E",b) ← (Y, falso);
                               enquanto Y'≠∅∧ ¬b faça
                                  ⟅ η FUNCIONÁRIO F:F∈E";
                                    (E",b) ← (E"-{F},F•nome-do-ge
                                       rente=G•nome-do-funcionário ⟆;
                               b ⌋então (G•nome-do-funcionário G•sa-
                                         lário) senão ∅ ⌋⌋ ;
                Z⌋
```

Note-se que, neste caso, P(x) de F3 do capítulo anterior é idêntico a verdadeiro e portanto não ocorre na conjunção.

5.3 - EXEMPLO DE IMPLEMENTAÇÃO DO OPERADOR DE ESCOLHA E MUDANÇA NAS ESTRUTURAS DE DADOS

Observando-se o programa acima, notamos que a implementação do operador de escolha η é, em ambos os casos, relativamente simples. De fato, note-se que os conjuntos envolvidos são finitos e sua cardinalidade diminui depois de cada aplicação de η; além disso, η nunca seleciona duas vezes o mesmo elemento, pois a cada nova passagem pela declaração que o utiliza, muda o conjunto de elementos dos quais um é escolhido, exatamente pela eliminação do elemento selecionado na passagem anterior.

Para exemplificarmos uma possível implementação do operador de escolha η, vamos supor que as relações estejam arquivadas de tal maneira que possam ser buscadas por meio de um índice, isto é, cada valor do índice está associado univocamente a uma n-pla da relação. Do ponto de vista de linguagens de programação, tudo se passa então como se cada relação fosse uma matriz ("array") de uma dimensão; cada elemento da matriz tem a estrutura dos tem a estrutura dos registros da relação. Vale notar que algumas linguagens da programação, como por exemplo PASCAL [27] e PL/1 permitem "arrays" de registros.

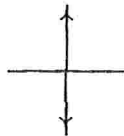
A transformação seguinte será baseada no esquema abaixo, onde conj t indica um tipo de conjunto com elementos de tipo t e |X| a cardinalidade do conjunto X; t [1:n] X declara X como sendo um "array" de uma dimensão, com limites

1 e n com cada elemento do tipo t.

(E39)

```
conj t X
   ⋮
┌ η t x:x∈X;
  X ← X-{x} ┘;
   ⋮
```

a) |X| = n
b) X implementado como X [i], i=1,2,...,n

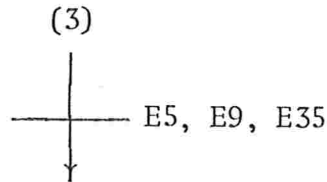


```
t [1:n] X;
   ⋮
┌ t x;
  η int i:1≤i≤n;
  x ← X[i];
  X[i] ← X[n];
  n ← n-1 ┘
```

Com esse esquema reduzimos o problema de escolher um elemento de um conjunto ao problema de escolher um valor para uma variável inteira i. Como no caso anterior, não há perigo do operador escolher o mesmo elemento pois também no caso da matriz, o elemento escolhido é eliminado em seguida.

A condição $X = \emptyset$ equivale a $|X| = 0$, isto é, a $n = 0$. Temos, então, expandindo o corpo de C_4, eliminando C_4', lembrando a implementação descrita antes de E35, supondo que card(X) seja uma função que retorne como valor a cardinalidade do conjunto X, supondo ainda que se possa escrever (e de alguma maneira executar) E ← X ou X ← E onde tem-se relação r X com a cardinalidade de X igual a n e r[1:n]E:

(3)



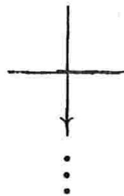

```
função C4 (EMPRESA E) relação (cadeia N, int SAL):
    {⌈int n;
     n ← card(E);
      ⌈int n';
       FUNCIONÁRIO [1:n] E';
       relação (cadeia N, int SAL) Z;
       (E',Z) ← (E,∅);
       n' ← n;
       enquanto n'≠0 faça
           ⌈⌈FUNCIONÁRIO G;
            η int i:1≤i≤n';
            G ← E'[i];
            E'[i] ← E'[n'];
            n' ← n'-1;
(4)         Z ← Z ∪ se ⌈int n";
                         FUNCIONÁRIO [1:n]E"; log b;
                         (E",b) ← (E, falso);
                         n" ← n;
                         enquanto n"≠∅∧ ¬b faça
                            {⌈FUNCIONÁRIO F;
                             η int j:1≤j≤n";
                             F ← E"[j];
                             E"[j] ← E"[n"];
                             n" ← n"-1;
                             b ← F•nome-do-gerente=G•nome-
                                           do-funcionário ⌋};
                        b⌋ então (G•nome-do-funcionário,
                                   G•salário) senão ∅⌋⌋;
       Z⌋}
```

Observe-se que fizemos algumas transformações adicionais sem justificá-las formalmente pois teríamos que fazer muitos passos intermediários para deduzi-las. Elas têm apenas a utilidade de introduzirem as novas variáveis n, n' e n'' e não alteram fundamentalmente e estrutura do programa.

Uma objeção à versão (4) do programa é que estamos misturando conjuntos com matrizes. A nossa intenção foi a de preservar a estrutura (ou, melhor, a falta dela) da relação dada (passada através do parâmetro E) e do resultado, como estruturas abstratas tipo conjunto (relação). Mais adiante veremos o impacto da restrição dessas estruturas a estruturas do tipo matriz. No momento, estamos mais interessados em apresentar uma possível solução para a implementação do operador η.

Examinando-se o primeiro operador η, verificamos que ele escolhe um inteiro arbitrário entre 1 e n'. Uma implementação possível seria então tomar sempre i = n'. Com isso teríamos:

(4)



⋮

(5)

```
┌ FUNCIONÁRIO G;
  int i;
  i ← n';
  G ← E'[i];
  E'[i] ← E'[n'];
  n' ← n'-1;
  ⋮
```

Aplicando o seguinte esquema (vide notação de E5)

onde eventualmente b e F são n-plas de mesmo número de elementos,

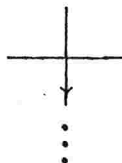
(5)

```
⋮
┌FUNCIONÁRIO G;
 int i;
 i ← n';
 G ← E'[n'];
 E'[n'] ← E'[n'];
 n' ← n'-1;
⋮
```

(6)

A variável i passa a não ser mais usada em lugar algum, e pode ser eliminada, bem como a 4ª atribuição (vide E24)

(6)

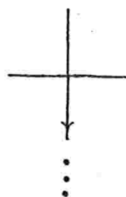


```
⋮
┌FUNCIONÁRIO G;
 G ← E'[n'];
 n' ← n'-1;
⋮
```

(7)

No caso de η int j de (4), poderíamos aplicar exatamente a mesma busca. No entanto, um exame do bloco {⌐... ⌙} mostra que a sua execução consiste essencialmente em varrer-se a cópia E" de E à procura de uma n-pla F tal que sua projeção em nome-do-gerente é igual a G•nome-do-funcionário. Ora, este último valor é conhecido no bloco em questão. Este fato pode ser usado para se imaginar uma busca mais eficiente em E". Se E" for ordenado segundo os valores do domínio nome-do-gerente, o operador η poderia ser implementado por meio de uma busca binária, isto é,

(7)



(8)

```
        ⋮
{⌐ FUNCIONÁRIO F;
   int inf,sup;j;
   inf ← 1;
   sup ← n";
   enquanto inf ≠ sup∧ ¬b faça
    ⌈j ← (inf + sup) div 2;
     F ← E"[j];
     se F•nome-do-gerente=G•nome-do-
           funcionário
        então b ← verdadeiro
        senão se F•nome-do-gerente>G•nome-do-
                    funcionário
                então sup ← j
                senão inf ← j⌋
                                        ⌙}
        ⋮
```

onde <u>div</u> é a divisão inteira, > uma comparação usando a sequência alfabética de caracteres e onde o algoritmo de busca binária deveria ter sido previamente provado.

Outros algoritmos de busca poderiam ser usados, por exemplo cálculo do índice ("hashing"), etc.

Damos abaixo uma versão final do algoritmo de consulta, usando as duas implementações vistas para os operadores η. Expressaremos a relação-parâmetro como uma matriz unidimensional, adicionando mais um parâmetro que indica o limite superior n da variação de seu índice. Em lugar do operador $\cup$ de (4), empregaremos uma função insere ([ ]X,p,<u>r</u>Y), onde X é uma matriz unidimensional com p elementos do tipo <u>r</u> inseridos até esta chamada; p = 0 indica que nenhum elemento foi ainda inserido em X; se Y é idêntico a um dos elementos de X, nada é feito, caso contrário, ou se p = 0, Y é inserido em X e p incrementado de 1. Note-se que o fato de se alterar n' e n'' não implica na alteração de E' e E'', o que se poderia ter concluído de (4) mas que fica óbvio em (7) e (8). Assim sendo, usaremos o próprio parâmetro E.

Note-se que sup de (8) está fazendo o papel de n'' de (4).

Finalmente, observe-se que o uso da busca binária nos blocos mais interiores exige que a matriz em que se busca o elemento procurado (no caso, G•nome-do-funcionário) esteja ordenada lexicograficamente, isto é, a matriz-parâmetro E deve ter essa ordenação.

(8)

```
função C4 (FUNCIONÁRIO []E,int n) [](cadeia N,int SAL),int:
   ⌈int n',m;
    (cadeia N, int SAL)[1:n]Z;
    <<a matriz-resultado não terá mais elementos do que a ma-
                                                    triz-parâmetro>>
    m ← 0;
    <<m indica o número de elementos da matriz-resultado>>
    n' ← n;
    enquanto n'≠0 faça
        ⌈FUNCIONÁRIO G;
        G ← E[n'];
        n' ← n'-1;
        se ⌈FUNCIONÁRIO F;
            int inf,sup,j; log b;
            (inf,sup,b) ← (1,n,falso)
            enquanto inf≠sup∧¬b faça
               ⌈j ← (inf+sup) div 2;
                F ← E[j];
                se F•nome-do-gerente=G•nome-do-funcionário
                   então b ← verdadeiro
                   senão se F•nome-do-gerente>G•nome-do-fun
                                                    cionário
                            então sup ← j
                            senão inf ← j ⌋⌋;
            b⌋ então insere ([]Z,m,(G•nome-do-funcionário,
                                                G•salário)) ⌋⌋;
    (Z,m)⌋
```

Este algoritmo é plenamente "executável", e pode ser facilmente convertido para uma das linguagens de programação usuais, restando somente implementar a função "insere". Essa inserção será otimizada no tempo se Z for constituída segundo algum processo de endereçamento (algoritmo de "hashing"); menos eficiente seria empregar-se novamente uma busca binária, já que a inserção nesse método exige um tempo adicional razoável. Supondo-se que nome-do-funcionário seja a chave de E, a inserção poderia ser realizada sequencialmente pois não haveria perigo de que o mesmo nome-do-funcionário aparecesse duas ou mais vezes na matriz resultado Z. Neste caso, poderíamos ter o seguinte procedimento de inserção:

```
procedimento insere (r[]P,m,r reg):
        ⌈m ← m+1;
         P[m] ← reg⌋
```

supondo-se r tivesse sido declarado como

registro (cadeia, int)r.

Se se dispusesse de uma linguagem em que tipos pudessem ser transmitidos como parâmetros (!) o procedimento insere poderia ser independente de r, recebendo-o como parâmetro.

É interessante notar-se uma consequência do método aqui exposto, qual seja a de se poder projetar as estruturas dos dados para aumentar a eficiência do processamento, durante a fase de desenvolvimento dos programas. Assim, decidindo-se por um algoritmo de busca binária, como o fizemos, ter-se-ia imposto uma ordenação da relação original passada pelo parâmetro E, e segundo o campo nome-do-funcionário.

A formulação que empregamos, de considerar todas as relações como conjuntos permitiu que, posteriormente, introduzíssemos modificações nas relações, associando a elas estruturas particulares, no caso matrizes unidimensionais de registros-ordenados segundo um de seus campos. Nesse sentido a representação inicial como conjuntos, que em essência é uma estrutura "pouco estruturada" dá uma grande liberdade para posteriores particularizações em estruturas mais convenientes para o processamento. Como a escolha dessas últimas deriva dos algoritmos deduzidos para as consultas que por sua vez altera estes últimos, temos aqui um exemplo de "projeto iterativo" de dados e de algoritmos, em que esses e aqueles são definidos no decorrer do desenvolvimento do programa, procurando-se com isso aumentar a eficiência de processamento. Essa eficiência foi exemplificada no sentido de se diminuir o tempo de execução; em outros casos, pode-se buscar a otimização do espaço de armazenamento requerido. Em parte fizemos também certas transformações nesse sentido, como foi o caso de se eliminar as relações temporárias (auxiliares) E' e E" de (4).

- ¤ -

CAPÍTULO 6

CONCLUSÕES

Nos capítulos anteriores fizemos uma introdução a um método de dedução formal de programas por meio de transformações, e aplicamos esse método a consultas a bases de dados formuladas em cálculos de predicados. Por meio de um exemplo, mostramos como esse método pode levar ao desenvolvimento de programas otimizados para execução das mencionadas consultas. Uma pergunta que surge é a seguinte: será este método viável em termos práticos, isto é, poderá uma empresa, utilizá-lo na implementação de suas consultas às bases de dados que ela mantém?

Observando-se os caminhos seguidos, logo se constata que o método não pode ser utilizado por pessoas que não tenham um bom nível matemático e um conhecimento razoável de programação, estruturas de dados, etc. Nota-se também que uma dedução feita nesses moldes é penosa e requer um tempo razoável. Na verdade, os passos iniciais poderiam ser automatizados. Tendo-se formulado a consulta na forma F8 ou alguma das antecedentes mais simples, um programa poderia fornecer a forma iterativa do algoritmo correspondente. A partir daí, entramos nas transformações que visam a implementação dos conjuntos em estruturas disponíveis, bem como a realiza

ção das buscas representadas pelo operador de escolha. Essas poderiam eventualmente ser automatizadas, através de um catálogo de transformações como o proposto em [4]. Esse catálogo poderia ser razoavelmente restrito, pois os tipos de estruturas que ocorrem nas bases de dados são relativamente limitados; os algoritmos de busca também podem ser reduzidos a um número não muito grande. No entanto, apesar dessas restrições, a escolha das transformações, das estruturas e das buscas dependeriam de um conhecimento que consideramos não disponível na maioria de programadores e analistas. Considerando-se a notória ineficiência dos sistemas ("packages") de bases de dados existentes no mercado, cremos que o método aqui exposto poderia representar um considerável ganho em consultas que se repetem constantemente e para as quais um programa "eficiente" seria altamente rentável. Nesse caso, a contratação de um cientista da computação ou de um matemático talvez venha a ser uma boa solução, permitindo a aplicação do método aqui exposto. Dessa maneira poderiam ser deduzidos programas corretos que efetuam as consultas mais comuns e críticas do sistema. Evidentemente uma linguagem comum de consulta poderia ser inicialmente empregada, até que os programas fossem deduzidos e testados (não se pode descartar um possível erro de dedução!).

As considerações acima levam, portanto, à seguinte resposta à nossa pergunta inicial: o método é viável em casos práticos particulares, na medida em que se dispõe de pessoal qualificado e se necessite otimizar certas consultas usadas com muita frequência.

Ampliações deste trabalho poderiam ser propostas nas seguintes direções:

1. Dedução de esquemas de transformações para linguagens

de consultas existentes derivadas da linguagem ALPHA de Codd, tais como QUEL [18] e SEQUEL [7]. De maneira geral, nossos esquemas englobam as construções tradicionais.

2. Dedução de esquemas de transformação para construções do tipo "agregados", como a cláusula "where" de SEQUEL [7].

3. Dedução de esquemas para linguagens de consultas baseadas em álgebra de relações [8], [9].

4. Implementação de programas que façam a qualificação automática de esquemas, e desenvolvimento de catálogo de transformações adequadas a consultas a bases de dados.

5. Estudo de transformações em nível de predicados, a fim de se obter uma otimização na busca já nesse nível.

6. Estudo de transformações ao nível da versão recursiva, antes de se eliminar a recursão.

- ¤ -

APÊNDICE

1 - PROVA DE E2

A prova abaixo é baseada em parte na de Gnatz [17]; omitimos detalhes das deduções para não sobrecarregarmos em demasia o texto e para obter-se melhor visão do conjunto:

E2:

(a) $C \Longrightarrow \forall \underline{t}'x(K(x) \Longrightarrow \exists \underline{t}u(Q(x,u))$

(b) $C \Longrightarrow \exists \underline{t}'z'R(z)$

$\eta \underline{ty}: P(y)$

⌈função $f(\underline{t}'x)\underline{t}$:
$\eta \underline{t}u: Q(x,u)$;
$f(\eta \underline{t}'z: R(z)$⌋.

onde $R(z) \equiv (C \Longrightarrow K(z)) \vee \forall \underline{t}u(Q(z,u) \Longrightarrow P(u))$.

Da pré-condição (b), usando o Axioma da Escolha $\vdash \exists x{:}P(x) \Longrightarrow P(\eta x{:}\ P(x))$,e o fato de que $(A \Longrightarrow B,\ B \Longrightarrow C) \vdash (A \Longrightarrow C)$:
(i):

(1) $\vdash C \Longrightarrow ((C \Longrightarrow K(\eta \underline{t}'z{:}R(z))) \wedge \forall \underline{t}u(Q(\eta t'z{:}R(z),u) \Longrightarrow P(u)))$

Do fato $A \Longrightarrow (B \wedge C) \vdash A \Longrightarrow B$ (e $\vdash A \Longrightarrow C$) bem como $A \Longrightarrow (A \Longrightarrow B) \vdash A \Longrightarrow B$ (pois $A \Longrightarrow (B \Longrightarrow C) \vdash A \wedge B \Longrightarrow C$)

(ii):

(2) $\vdash C \Longrightarrow K(\eta \underline{t}'z:R(z))$

(3) $\vdash C \Longrightarrow \forall \underline{t}u(Q(\eta \underline{t}'z:R(z),u) \Longrightarrow P(u))$

Da pré-condição (a), como $\forall \underline{t}a(P(a)) \Longrightarrow P(b)$ (iii), usando (i) acima:

(4) $\vdash C \Longrightarrow (K(v) \Longrightarrow \exists \underline{t}u(Q(v,u)))$

De (ii) e aplicando o Axioma de Escolha

(5) $\vdash C \wedge K(v) \Longrightarrow Q(v,\eta \underline{t}u:Q(v,u))$

Usando uma modificação da notação λ de Church [cf. 22], temos $A(z) \equiv \ulcorner(\underline{t}'x)\underline{t}: A(x)\lrcorner(z)$ onde z é o tipo $\underline{t}'$ e A assume valores em $\underline{t}$ (iv):

(6) $\vdash C \wedge K(v) \Longrightarrow Q(v,\ulcorner(\underline{t}'x)\underline{t}: \eta\underline{t}u: Q(x,u)\lrcorner(v))$

Como v é livre, temos:

(7) $\vdash C \wedge K(\eta\underline{t}'z:R(z)) \Longrightarrow Q(\eta\underline{t}'z:R(z),\ulcorner(\underline{t}'x)\underline{t}:\eta\underline{t}u:Q(x,u)\lrcorner(\eta\underline{t}'z:R(z)))$

Mas $\vdash A \Longrightarrow B$ e $\vdash A \wedge B \Longrightarrow C$ levam a $\vdash A \Longrightarrow C$, portanto de (2) e (7):

(8) $\vdash C \Longrightarrow Q(\eta\underline{t}'z:R(z),\ulcorner(\underline{t}'x)\underline{t}:\eta\underline{t}u:Q(x,u)\lrcorner(\eta\underline{t}'z:R(z)))$

Usando (iii), podemos particularizar (3) para um certo v em lugar de u, e aplicando (i):

(9) $\vdash C \Longrightarrow (Q(\eta\underline{t}'z:R(z),v) \Longrightarrow P(v))$

Usando novamente (iv) substitui-se v de (9) como em (5), e aplicando (i):

(10) $\vdash C \Longrightarrow (Q(\eta\underline{t}'z:R(z),\ulcorner(\underline{t}'x)\underline{t}':\eta\underline{t}u:Q(x,u)\lrcorner(\eta\underline{t}'x:R(z)))$

$\Longrightarrow P(\ulcorner(\underline{t}'x)\underline{t}:\eta\underline{t}u:Q(x,u)\lrcorner(\eta\underline{t}'z:R(z)))$

Como $\vdash A \Longrightarrow B$ e $\vdash A \Longrightarrow (B \Longrightarrow C)$ levam a $\vdash A \Longrightarrow C$ (iv), de (8) e (10):

(11) $\vdash C \Longrightarrow P(\ulcorner(\underline{t}'\ x)\underline{t}:\eta\underline{t}u:Q(x,u)\lrcorner)(\eta\underline{t}'z:R(z))$

Pela definição de procedimento:

(12) $\vdash P(\ulcorner(\underline{t}'x)\underline{t}:\eta\underline{t}u:Q(x,u)\lrcorner)(\eta\underline{t}'z:R(z)) \Longrightarrow$

$$P(\ulcorner \underline{\text{função}}\ f(\underline{t}'x)\underline{t}:\eta\underline{t}u:Q(x,u);f(\eta\underline{t}'z:R(z))\lrcorner)$$

Portanto, por (i):

(13) $\vdash C \Longrightarrow P(\ulcorner(\underline{\text{função}}\ f(\underline{t}'x)\underline{t}:\eta\underline{t}u:Q(x,u);f(\eta\underline{t}'z:R(z))\lrcorner)$

Como o antecedente de E2 é $\eta\underline{t}y:P(y)$, isto é, satisfaz ao programa P, temos (Ver pg. 2):

(14) $\vdash C \Longrightarrow P(\eta\underline{t}y:P(y))$

Comparando (13) e (14), temos o esquema procurado.

#

2 - PROVA DE E8 (Cf. Gnatz [17])

$$\left.\begin{array}{l} a)\ \vdash C \Longrightarrow \exists\underline{t}x(Q(x)) \\ \\ b)\ \vdash C \Longrightarrow (Q(x) \Longrightarrow P(x)) \end{array}\right\} \begin{array}{c} \eta\underline{t}x:P(x) \\ \hline \downarrow \\ \eta\underline{t}x:Q(x) \end{array}$$

Aplicando o Axioma da Escolha a (a) e usando (i) da prova de E2 acima:

(1) $\vdash C \Longrightarrow Q(\eta\underline{t}x:Q(x))$

Substituindo x de (b) pelo argumento de Q em (1) e novamente empregando (vi) da prova de E2:

(2) $\vdash C \Longrightarrow P(\eta\underline{t}x:Q(x))$

Como o antecedente de E8 é $\eta\underline{t}x:P(x)$ temos

(3) $\vdash C \Longrightarrow P(\eta\underline{t}x:P(x))$

donde E8 está provada. #

3 - PROVA DE E10

No esquema abaixo $S_{a,b}$ indica que a e b ocorrem no comando (ou expressão) S; como anteriormente, S_b^a indica S reescrito substituindo-se todas as ocorrências de a por b; x' pode ser uma expressão em x. Evidentemente, $\underline{t}$' deve ser do tipo lógico. Adotamos uma notação um pouco diferente de E10 para maior clareza. Queremos provar:

(1)

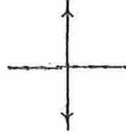
$$\ulcorner \underline{\text{função}}\ f(\underline{t}x)\underline{t}' : S_{x,f};$$
$$\vdots$$
$$\neg f(y)$$
$$\vdots$$
$$\lrcorner$$

$$\Updownarrow$$

(2)

$$\ulcorner \underline{\text{função}}\ f(\underline{t}x)\underline{t}' : S_{x,f};$$
$$\underline{\text{função}}\ f'(\underline{t}x)\underline{t}' : \neg \left[S_{x,f}\right]^{f}_{\neg f'};$$
$$\vdots$$
$$f'(y)$$
$$\vdots$$
$$\lrcorner$$

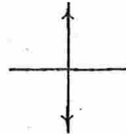
PROVA



(3)

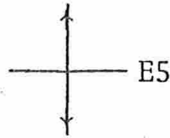
$\ulcorner$ função f(tx)t':$S_{x,f}$;
função f'(tx)t':$\neg S_{x,f}$;
⋮
$\neg f(y)$
⋮
$\lrcorner$

Como por E5, $\neg f(y) \equiv \neg \left[S_{x,f} \right]_y^x \equiv f'(y)$.



(4)

$\ulcorner$ função f(tx)t':$S_{x,f}$;
função f'(tx)t':$\neg S_{x,f}$;
⋮
f'(y)
⋮
$\lrcorner$



$$\ulcorner \underline{\text{função}}\ f(\underline{t}x)\underline{t}':S_{x,f};$$

$$\underline{\text{função}}\ f'(\underline{t}x)\underline{t}':\neg\left[S_{x,f}\right]^{\neg f(x')}_{[S_{x,f}]^{x}_{x'}};$$

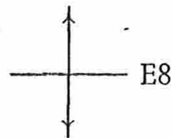
(5)

⋮

f'(y)

⋮

$\lrcorner$



$$\ulcorner \underline{\text{função}}\ f(\underline{t}x)\underline{t}':S_{x,f};$$

$$\underline{\text{função}}\ f'(\underline{t}x)\underline{t}':\neg\left[S_{x,f}\right]^{\neg f(x')}_{\neg\neg[S_{x,f}]^{x}_{x'}};$$

(6)

⋮

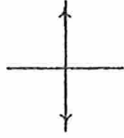
f'(y)

⋮

$\lrcorner$

Mas, por E5, $\neg\left[S_{x,f}\right]^{x}_{x'} \equiv f'(x')$ de (3):

(6)



⌈ <u>função</u> f(<u>t</u>x)<u>t</u>':$S_{x,f(x')}$;

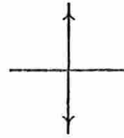
<u>função</u> f'(<u>t</u>x)<u>t</u>':$\neg \left[S_{x,f} \right]^{f(x')}_{\neg f'(x')}$

⋮

f'(y)

⋮

⌋



(2)

#

- ¤ -

REFERÊNCIAS

[1] - Bauer,F.L., "Variables Considered Harmful und andere Bemerkungen zur Programmierung", T.U.München,Inst.für Informatik,*Bericht Nr. 7519* (Okt.1975).

[2] - Bauer,F.L.; Partsch,H.;Pepper,P. & Wössner,H.,"Techniques for Program Development", T.U.München,Inst.für Informatik, *Interner Bericht* (Sept.1976).

[3] - Bauer,F.L. et al.,"Notes on the project CIP: towards a wide spectrum language to support program development by transformations", T.U.München, *TUM-INFO-7722* (July 1977).

[4] - Bauer,F.L.;Partsch,H.;Pepper,P. & Wössner,H.,"Notes on the project CIP: outline of a transformation system", T. U. München, *TUM-INFO-7729* (July 1977).

[5] - Burstall,R.M.& Darlington,J.,"Some transformations for developing recursive programs", *Proc. of 1975 International Conference on Reliable Software,* Los Angeles (1975), pp. 465-472.

[6] - Burstall,R.M.& Darlington,J.,"A Transformation System for Developing Recursive Programs", *Journal ACM 24,1* (Jan.1977)pp.44-67.

[7] - Chamberlin,D.D. et al.,"SEQUEL 2: A unified Approach to Data Definition, Manipulation and Control", *IBM Journal of Research and Development 21,6* (Nov.1976),pp. 560-575.

[8] - Codd,E.F., "A relational model of data for large shared data bank", *Comm. ACM 13,6* (June 1970), pp. 377-387.

[9] - Codd,E.F., "Relational completeness of Data Base sublanguages", in Rustin,R.(ed.),*Data Base System*, Courant Computer Science Symposium 6, Prentice-Hall, Engewood Cliffs (1972).

[10] - Codd,E.F., "Further normalization of the Data Base relational model", idem [9].

[11] - Conway,M.E.,"Design of a separable transition diagram compiler", *Comm. ACM 6,7* (July 1963), pp. 396-408.

[12] - Dahl,O.J.;Dijkstra,E. & Hoare,C.A.R., *Structured Programming*,APIC studies on data processing Nº 8, London Academic Press, London (1972).

[13] - Dijkstra,E.W., "Go to statement considered harmful", Letter to the Editor, *Comm. ACM 11,3* (March 1968), pp. 147-148.

[14] - Dijkstra,E.W., "Guarded Commands,Nondeterminancy and Formal Derivation of Programs", *Comm. ACM 18,8* (Aug.1975),pp.453-457.

[15] - Floyd,R.,"Assigning Meanings to Programs", in Schwarz,J.T. (ed.), *Mathematical Aspects of Computer Science*, Amer. Math. Soc., Providence, RI (1967), pp. 19-32.

[16] - Gnatz,R., "Interactive Program Development", T.U. München, Inst. für Informatik, *Interner Bericht*, (Aug.1976).

[17] - Gnatz,R., "Deduktive Programentwicklung", T.U.München, Inst. für Informatik, *Interner Bericht Nr. 7631* (Nov.1976).

[18] - Held,G.D.;Stonebraker,M. & Wong,E., "INGRES — a relational Data Base System", *Proc. AFIPS National Computer Conf.*,AFIP Press, Anaheim (May 1975), pp. 409-416.

[19] - Hoare,C.A.R.,"An axiomatic Basis for Computer Programming",*Comm. ACM, 12, 10* (Oct.1969, pp. 576-580,583.

[20] - Lapyda,R., *Introdução aos Modelos de Sistemas Administradores de Bases de Dados*, Dissertação de Mestrado, IME-USP, São Paulo, (maio de 1977).

[21] - Manna,Z., *Mathematical Theory of Computation*, McGraw-Hill,N.York (1974).

[22] - McCarthy,J., "A basis for a mathematical theory of computation", in *Computer Programming and Formal Systems*, Brafort, P. & Hirschberg,D. (Eds.), North Holland, Amsterdam (1967).

[23] - Naur,P. et alli, "Revised Report on the Algorithmic Language ALGOL 60", *Comm. ACM 6, 1* (Jan.1963), pp. 1-17.

[24] - Partsch,M. & Pepper,P., *Program transformations on different levels of programming*, T.U.München, TUM-INFO-7715 (Juni 1977).

[25] - Schwartz,J.T. "On programming: an interim report on the SETL project, Installement I:Generalities",New York Univ.,Courant Inst. of Mathematical Sciences, Computer Science Dept. (1973).

[26] - van Wijngaarden,A.et alli,"Revised Report on the Algorithmic Language ALGOL 68", *Acta Informatica, 5, 1-3* (1975),pp.1-236.

[27] - Wirth,N., "The Programming Language PASCAL", *Acta Informatica,1, 1*, (1971), pp. 35-63.

[28] - Wirth,N., "Program Development by Stepwise refinement, *Comm. ACM 14, 4* (April 1971), pp. 221-227.

[29] - Wirth,N., *Systematic Programming*, Prentice Hall,Englewood Cliffs (1973).

[30] - Wirth,N. & Hoare,C.A.R., "A contribution to the Development of ALGOL", *Comm. ACM 9, 6* (June 1966), pp. 413-441.

- ¤ -

ÍNDICES DOS ESQUEMAS DE TRANSFORMAÇÃO E DAS FORMAS DE CONSULTAS

ESQUEMA	PAG.
E1	4
E2	5
E3	6
E4	9
E5	10
E6	17
E7	19
E8	20
E9	24
E10	25
E11	26
E12	27
E13	28
E14	28
E15	29
E16	58
E17	64
E18	65
E19	68
E20	68
E21	72
E22	72
E23	76
E24	76
E25	77
E26	78
E27	79
E28	81
E29	82
E30	84
E31	84
E32	86
E33	87
E34	88
E35	88
E36	90
E37	91
E38	91
E39	95
E40	98

FORMA	PAG.
F1	57
F2	68
F3	69
F4	81
F5	84
F6	86
F7	87
F8	90